Broadview®
www.broadview.com.cn

系统深入
案例丰富

App
后台开发运维和架构实践

曾健生 编著

本书像一本《十八般武器入门》，告诉读者如何用“正确的方式”使用各种已有的工具，并为读者呈现一幅包括技术选型、后台搭建、性能优化、运维实践、架构设计在内的App后台开发蓝图。

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

曾健生，曾任职于广州市赢靖信息科技有限公司，负责社交App后台研发。目前就职Bmob后端云从事云服务方面的研发工作。

读者在阅读本书的过程中有任何问题和建议，请通过以下方式联系作者。

●微信公众号：

app后端（微信号：appbackend）

●博客：

http://blog.csdn.net/newjueqi

App 后台开发运维和架构实践

曾健生 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

《App 后台开发运维和架构实践》通过阐述移动互联网中 App 后台开发的特点，梳理了 App 后台开发中会遇到的各个技术点，给出了生产环境常用软件的实战运维经验总结，剖析了常见 App 后台技术架构设计，为读者呈现一幅包括技术选型、后台搭建、性能优化、运维实践、架构设计的 App 后台开发蓝图。

本书的目标读者是对技术感兴趣的产品经理、刚入行的 App 后台开发人员，以及从传统软件行业转向 App 后台开发的技术人员。

图书在版编目（CIP）数据

App 后台开发运维和架构实践 / 曾健生编著. —北京：电子工业出版社，2016.5
ISBN 978-7-121-28380-2

Ⅰ. ①A… Ⅱ. ①曾… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2016）第 056285 号

责任编辑：付　睿
印　　刷：北京天宇星印刷厂
装　　订：北京天宇星印刷厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：17.5　　字数：387.5 千字
版　　次：2016 年 5 月第 1 版
印　　次：2016 年 9 月第 3 次印刷
定　　价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

推荐序

十八般兵器入门

软件开发工具的成长速度远远超过开发人员的成长速度，这是现实。

每个月，甚至每天，我们都可以见到新的类库、框架、工具、语言。它们或者极大地降低了开发的成本，或者极大地提升了开发的效率。

随之而来的问题就是，如何让开发人员妥善地运用好这些工具？

曾经有好几个做开发的同事跟我说：“写软件谁不会？从网上找些开源的类和项目来改改就是了”。目前也确实充斥着这种“改改就是”的工作思维。介绍某个类库和项目如何安装和调用的文章俯拾皆是。

但是每种工具究竟有什么优缺点？适合用来解决什么问题？需要以怎样的方式来解决问题？因此还需要做怎样的权衡？这样的问题基本没有人考虑，但是，它们又很重要。

这有点像练武。斧钺钩叉、刀枪剑戟，制造手艺日渐精良；可是习武的人心里没有分寸，该劈的时候提了枪，该刺的时候抡了斧…… 耍几下花把式还算足够，真正打起来哪能取胜？

我在面试新人中经常提的问题是：NoSQL 分为哪几种类型？每种类型的典型代表和应用场景是什么？要知道，用过 MongoDB 和 Redis 的人比比皆是，但回答不上这几个问题的也大有人在。这样的候选人，我是决计不敢要的。稍加不注意，他们就可能用 Redis 存文档，用 MongoDB 做复杂运算，甚至“大胆”用 Redis 来替换数据库以解决性能瓶颈——噢，你说起“事务”，Redis 也是有“事务”的呀。

对这样的人，你真让他们去研究 MongoDB 或者 Redis，让他们去通读一本专著，似乎有点划不来，他们自己也没有那么多耐心。所以我常想，有没有一本“薄而广”的书，专注于开

拓大家的眼界，并教导大家用“正确的方式”来使用各种已有的工具。如今我们虽然有海量的框架和类库，有突飞猛进的云服务，但是只要没有掌握“正确的应用方式”，就无法保证“把事情做对”，就只能开发出某几个方面明显不及格的产品。

所以当我看到曾健生编著的《App 后台开发运维和架构实践》时，不由得眼前一亮。这正是一本“薄而广”的书，它绝不像《打狗棍法》或者《辟邪剑法》，不负责地教你把某门特别的武功练到极致，而更像《十八般兵器入门》，告诉你刀枪剑戟各适合什么场景，使用时有什么注意事项。典型的例子如关于 Redis 的部分，它讲的不是 Redis 如何安装，如何调用，而是结合发送短信、社交关系计算等典型应用场景讲解 Redis 的使用方式，并介绍在 Redis 提供内建集群之前，市面上的各种集群的方案和优劣。这样虽然只有一章的篇幅，但为普通开发人员提供的价值绝对要比两三本 Redis 专著都要大，而且读起来更有效率。

古代练武的人，未必人人都要做武林高手，许多人只是把十八般兵器都耍熟，就已经足够防贼、保平安，受益匪浅了。同样的道理，对今天的 App 后台开发人员来说，把基本的点都踩到、踩准，把系统拎到及格线以上，避免明显的缺憾，对大多数场景来说，已经是意义重大了。

余晟

软件开发老兵，微信公众号“余晟以为”

前言

笔者在 2012 年从开发电子商务网站转向了开发 App 后台，当时在一家做社交 App 的创业公司里工作，笔者和搭档都没有任何从事移动互联网开发的经验，不清楚 App 后台怎么架构，只能摸着石头过河，那时网络上只有一些零散的资料，当遇到问题时只能不断地摸索和思考，来找到解决问题的方法。

在从事开发 App 后台接近 4 年的时间中，笔者参与了两款社交 App 的开发，现在就职于 bmob，从事云后台服务的研发工作，慢慢地对 App 后台的架构有了一些体会。

从 2013 年年底开始，笔者把工作笔记发表在 CSDN 博客专栏“App 后端技术架构”，陆陆续续收到了很多网友的反馈，后来在 QQ 里面接触了很多刚刚从事开发 App 后台的开发者、找技术合伙人的创业者，在聊天的过程中，发现很多基本的问题被不停地问，例如：

- 队列有什么用？
- Redis 的应用场景有哪些？
- 怎么保证通信的安全性？

看着对未来无限向往的同行也在重复着本人当初经历过的迷茫，也在纠结着这些技术问题，在网络上，没找到一本系统讲述 App 后台架构的书籍，能搜索到的不是“高大尚”公司的解决方案，就是针对一个技术点很详细的讲解。

于是在网友的鼓励下，笔者决定把本人所掌握的开发 App 后台的知识系统地讲解一次（当时计划是写 30 篇左右的文章），笔者利用业余的时间陆陆续续地写成文章发表在 CSDN 博客专栏“App 后端技术架构”，这也是本书前 3 章的初稿。

后来在博文视点的付睿编辑帮助下，笔者有机会把自身所学习的知识再系统地整理一次，以书籍的形式展现给各位读者，其中武小凤也参与了本书的编写工作。希望本书能够帮助更多的刚进入 App 后台开发的朋友们，以及对 App 后台技术感兴趣的产品经理和 Android、iOS 开发者。

因为本书的读者定位主要是 App 后台的初学者，因此笔者尽量以图文并茂的方式给读者介绍 App 后台各方面的技术。

笔者一向推崇的架构原则是，“尽量使用成熟可靠的云服务和开源软件，自身只专注于业务逻辑”，对于某项具体的技术必须掌握下面两点。

- 技术的应用场景。
- 技术的基本原理。

掌握上面两点后就能对这项技术有了基本的了解，在此基础上技术选型的优先级如下。

- 云服务。
- 开源软件。
- 自主研发。

通过上面的措施能在最大程度上减轻技术人员的额外研发负担，让自身的精力更加专注于业务。

至于某项技术的其他方面，例如怎么部署、开源软件的深度剖析等，笔者并不打算在本书中做深入的阐述，读者如果感兴趣可自行深入学习。笔者希望在本书中能给读者描绘一个 App 后台开发的蓝图。

本书主要分为 4 部分。

第 1~3 章：App 后台常用技术的讲解。

第 4~8 章：App 后台常用软件的运维和相关原理。

第 9 章：4 种类型 App 的后台架构。

第 10 章：App 后台架构的知识。

致谢

感谢 ekeo 的各位同事，和你们一起奋斗的日子终生难忘，虽然项目失败了，这段经历使笔者迅速成长，没有这段经历就不会有本书的诞生。

感谢 Bmob 平台的各位同事，特别是 CEO 何少岳博士、后台同事魏文生，你们身上对技术的热爱和对用户的热情使本人终生难忘。

感谢网络上的各位网友，你们的支持和鼓励是本人创作的原动力。

感谢网络上的众多技术分享者，你们的分享精神和分享的知识使笔者获益良多，同时也使笔者真切地明白“分享越多，获益越多”。

感谢《大型网站技术架构核心原理与案例分析》的作者李智慧，看完您的书后受用终生。

感谢博文视点的付睿编辑，没有她就没有本书的诞生，在本书编写的过程中，付睿编辑给了笔者很多细致的指导。同时也感谢博文视点的许多工作人员为本书最终出版所付出的努力。

感谢父母的养育之恩。

感谢亲爱的妻子，自从结缘于豆瓣书评以来，一路有您，倍感幸福。

目录

推荐序 3

前言 5

致谢 7

目录 8

第 1 章　App 后台入门 16

1.1　App 后台的功能 16
1.2　App 后台架构 17
1.3　App 和 App 后台的通信 19
1.4　App 后台和 Web 后端的区别 22
1.5　选择服务器 23
1.6　选择编程语言 24
1.7　快速入门新技术 25
1.7.1　思维模式 25
1.7.2　4 种快速入门新技术的方法 25
1.8　App 是怎样炼成的 26
1.8.1　项目启动阶段 26
1.8.2　研发阶段 28
1.8.3　测试阶段 29
1.8.4　正式推出阶段 29

1.9 最适合 App 的开发模式——敏捷开发 30
1.9.1 Sprint 计划会议 31
1.9.2 日常开发 32
1.9.3 每日例会 33
1.9.4 测试和修复 Bug 33
1.9.5 评审会议 34
1.9.6 回顾会议 34
1.9.7 及时反馈 34
1.9.8 总结 34

第 2 章 App 后台基础技术 35

2.1 从 App 业务逻辑中提炼 API 接口 35
2.1.1 业务逻辑思维导图 36
2.1.2 功能—业务逻辑思维导图 37
2.1.3 基本功能模块关系 40
2.1.4 功能模块接口 UML（设计出 API） 41
2.1.5 编写在线 API 测试文档 42
2.1.6 设计稿标注 API 45
2.2 设计 API 的要点 46
2.3 如何选择合适的数据库产品 50
2.3.1 Redis，MongoDB，MySQL 读写数据的区别 50
2.3.2 Redis，MongoDB，MySQL 查找数据的区别 50
2.3.3 Redis，MongoDB，MySQL 适用场景 51
2.4 如何选择消息队列软件 52
2.4.1 为什么要用消息队列？ 52
2.4.2 消息队列的工作流程 53
2.4.3 常见的一些消息队列产品 54
2.5 使用分布式服务实现业务的复用 54
2.5.1 巨无霸系统的危害 55
2.5.2 远程服务的优点 56
2.5.3 远程服务的实现 56
2.6 搜索技术入门 59

2.6.1 一个简单的搜索例子 59
2.6.2 搜索技术的基本原理 60
2.6.3 常见的开源搜索软件介绍 62
2.7 定时任务 65
2.7.1 Linux 定时任务 Crontab 65
2.7.2 在后台轻松管理各种各样的定时任务 66

第 3 章 App 后台核心技术 68

3.1 用户验证方案 68
3.1.1 使用 HTTPS 协议 68
3.1.2 基本的用户登录方案 69
3.2 App 通信安全 72
3.2.1 URL 签名 72
3.2.2 AES 对称加密 74
3.2.3 更进一步的通信安全 77
3.3 短信服务 78
3.3.1 App 后台发送短信简介 78
3.3.2 选择短信平台 78
3.3.3 建立可靠的短信服务 79
3.4 处理表情的一些技巧 80
3.4.1 表情在 MySQL 的存储 80
3.4.2 当文字中夹带表情的处理 80
3.4.3 Openfire 中发送表情引起连接断开的问题 81
3.5 高效更新数据 82
3.5.1 内容的推拉 83
3.5.2 数据增量更新策略 84
3.6 图片处理 90
3.7 视频处理 91
3.7.1 FFmpeg 简介 91
3.7.2 后台调用 FFmpeg 的功能 92
3.8 获取 APK 和 IPA 文件里的资源 94
3.8.1 Android 的 APK 文件 94

3.8.2 iOS 的 IPA 文件 96
3.9 文件系统 98
3.9.1 文件云存储服务 99
3.9.2 架设文件系统 99
3.10 ELK 日志分析平台 101
3.10.1 基本模块 101
3.10.2 日志分析流程 102
3.11 Docker 构建一致的开发环境 103
3.11.1 Docker 原理 103
3.11.2 搭建一致的开发环境 104

第 4 章 Linux——App 后台应用最广泛的系统 107

4.1 基本的系统优化 107
4.1.1 开机自启动服务优化 107
4.1.2 增大文件描述符 109
4.2 常用的命令 110
4.2.1 全面了解系统资源情况——top 110
4.2.2 显示进程状态——ps 115
4.2.3 查看网络相关信息——netstat 116
4.2.4 查看某个进程打开的所有文件——lsof 118
4.2.5 跟踪数据到达主机所经路由——traceroute 119
4.2.6 文件下载/上传工具——“ssh secure shell client”和“lrzsz” 119
4.2.7 查看程序的依赖库——LD_DEBUG 121
4.2.8 进程管理利器——superivisor 122
4.3 故障案例分析 125

第 5 章 Nginx——App 后台 HTTP 服务的利器 126

5.1 简介 126
5.2 基本原理 127
5.2.1 工作模型 127
5.2.2 进程解析 128
5.3 常用配置 129

5.3.1 Nginx 的全局配置 130
5.3.2 event 配置 130
5.3.3 http 配置 131
5.3.4 负载均衡配置 133
5.3.5 server 虚拟主机配置 134
5.3.6 location 配置 134
5.3.7 HTTPS 的配置 135
5.3.8 下载 App 的配置 136
5.3.9 生产环境中修改配置的良好习惯 136
5.4 性能统计 136
5.5 实现负载均衡的方案 137
5.6 用 Nginx 处理业务逻辑 139

第 6 章 MySQL——App 后台最常用的数据库 140

6.1 基本架构 140
6.2 选择版本 141
6.3 配置文件详解 142
6.4 软件优化 144
6.4.1 正确使用 MyISAM 和 InnoDB 存储引擎 144
6.4.2 正确使用索引 145
6.4.3 避免使用 select * 146
6.4.4 字段尽可能地设置为 NOT NULL 146
6.5 硬件优化 147
6.5.1 增加物理内存 147
6.5.2 增加应用缓存 147
6.5.3 用固态硬盘代替机械硬盘 148
6.5.4 SSD 硬盘+SATA 硬盘混合存储方案 149
6.6 架构优化 149
6.6.1 分表 150
6.6.2 读写分离 151
6.6.3 分库 153
6.7 SQL 慢查询分析 156

6.8 云数据库简介 157
6.9 灵活的存储结构 158
6.10 故障排除案例 159

第 7 章 Redis——App 后台高性能的缓存系统 160

7.1 Redis 简介 160
7.2 Redis 的常用数据结构及应用场景 161
7.2.1 string——存储简单的数据 162
7.2.2 hash——存储对象的数据 163
7.2.3 list——模拟队列操作 165
7.2.4 set——无序且不重复的元素集合 167
7.2.5 sorted set——有序且不重复的元素集合 168
7.3 内存优化 170
7.3.1 监控内存使用的状况 170
7.3.2 优化存储结构 170
7.3.3 限制使用的最大内存 172
7.3.4 设置过期时间 172
7.4 集群 174
7.4.1 客户端分片 174
7.4.2 Twemproxy 175
7.4.3 Codis 176
7.4.4 Redis 3.0 集群 179
7.4.5 云服务器上的集群服务 180
7.5 持久化 180
7.5.1 RDB 181
7.5.2 AOF 182
7.6 故障排除案例 184

第 8 章 MongoDB——App 后台新兴的数据库 185

8.1 简介 185
8.2 核心机制解析 186
8.2.1 MMAP（内存文件映射） 186

8.2.2 Journal 日志 187
8.3 入门 187
8.3.1 基本操作 188
8.3.2 数组操作 190
8.3.3 实例演示 MySQL 和 MongoDB 设计数据库的区别 191
8.4 高可用集群 195
8.4.1 主从 195
8.4.2 副本集 196
8.4.3 分片 198
8.5 LBS——地理位置查询 200
8.6 MongoDB 3.0 版本的改进 205
8.6.1 灵活的存储架构 206
8.6.2 性能提升 7~10 倍 206
8.6.3 存储空间最多减少 80% 207
8.6.4 运维成本最多降低 95% 207

第 9 章 App 后台架构剖析 208

9.1 聊天 App 后台架构 208
9.1.1 移动互联网的网络特性 209
9.1.2 协议 212
9.1.3 整体架构 218
9.2 社交 App 后台架构 221
9.2.1 基本表结构 222
9.2.2 推拉模式 223
9.2.3 数据库架构的演进 225
9.2.4 缓存架构的演进 229
9.3 LBS App 后台架构 234
9.3.1 地理坐标详解 235
9.3.2 查找附近的人 236
9.3.3 基于 MongoDB 的 LBS 后台架构演进 240
9.4 推送服务器后台架构 242
9.4.1 Android 推送 242

9.4.2 iOS 推送 248
9.5 获得更多 App 后台架构资料 252

第 10 章 App 后台架构的演进 255

10.1 架构的核心要素 255
10.1.1 高性能 256
10.1.2 高可用 258
10.1.3 可伸缩 261
10.1.4 可扩展 262
10.1.5 安全性 262
10.2 架构选型的要点 262
10.2.1 用成熟稳定的开源软件 263
10.2.2 尽可能使用云服务 264
10.3 架构的演进 268
10.3.1 单机部署 269
10.3.2 分布式部署 275
10.3.3 服务化 277
10.4 架构的特点 279
10.4.1 每个 App 的后台架构不会完全一样 279
10.4.2 架构的演进是由业务驱动的 279
10.4.3 架构不是为了炫耀技术 280

第 1 章
App 后台入门

在本章中，笔者从 App 后台的基础工作出发，帮助读者理清 App 后台和传统网站后端的区别，帮助开发者进行技术选型，并描述企业中 App 开发的整体流程，使开发者能对 App 后台的工作有初步的认识。

1.1 App 后台的功能

App 后台，也称为 App 后端，称呼不一样，但指的是同一个东西。

笔者一直都以为 App 后台的功能不用解释，但在网络上，准备用 App 创业的网友（是从传统行业过来的）问过这个问题，笔者就以 App 后台的两个主要功能简单地介绍一下。

注意： App 后台没有明确的定义，所涉及的范围也广，所以笔者解释的时候只选取 App 后台的两个主要的功能解析。为了保证通俗易懂，相关的概念会牺牲一定程度的准确性。如果已经了解 App 后台功能的读者，可跳过本节。

场景一：

用户 a 平时喜欢用音乐 App 听歌，音乐 App 保存了他平时最喜欢听的歌曲列表。

如果歌曲列表只保存在手机上，万一其手机被盗，就算买了一部新的手机回来，那歌曲列表也会丢失。

为了解决这个问题，其中一个办法就是音乐 App 把歌曲列表的数据放在远处的一台机器上，当用户 a 买了新手机后，把放在远处的机器上的歌曲列表重新下载到音乐 App 上就行。

场景二：

相信很多读者都有寄快递的经历。

假设有两个用户 a 和 b，当 a 向 b 寄一样东西的时候，会找快递员，把东西寄给 b。

用户 a 查看物流可以看到类似这样的描述“到达 xx 中转站”。通过中转站，东西就到达 b 的手上。

两个 App 之前传输信息的流程也相似：假设 App 上的用户 a 需要向用户 b 发信息，这条信息也需要经过 App 后台这个中转站，才能到达用户 b。

总结：

从以上的场景 1 和场景 2 可总结 App 后台的两个重要作用。

- 远程存储数据。
- 消息中转。

1.2　App 后台架构

App 后台架构，一个听起来“高大尚”的名字，很多读者听到这个词语感觉很迷茫，不明白架构具体是指什么？“App 后台应该怎样架构”这个问题笔者在 QQ 群被问了无数次。通过阅读本节，根据笔者提出的一个初级的通用架构设计框架，帮读者踏入架构的大门！

在百度百科中对架构的定义是：网站架构，一般认为是根据客户需求分析的结果，准确定位网站目标群体，设定网站整体架构，规划、设计网站栏目及其内容，制定网站开发流程及顺序，以最大限度地进行高效资源分配与管理的设计。

新手看到上面对架构的解析会被搞晕。

笔者根据自身的开发经验，为了帮助新手快速入门，特地提炼出一个初级的通用架构设计框架。

有什么业务？

遇到什么问题？

有什么可行的技术解决方案？

掌握了以上的架构设计的框架，有什么好处？

1. 不怕被别人的架构文章搞晕

当在网络上看到别人分享的架构文章时，套用这个架构的框架，问自己这 3 个问题：作者

是在什么业务逻辑遇到哪些问题，采用了哪些技术解决方案。通过这个框架，能帮助读者快速提炼出别人架构的核心点，掌握这个架构的精髓。

2. 能快速地整理合适的架构

当设计 App 后台的架构时，根据以上的架构框架，采用下面 4 点设计 App 架构。

（1）根据 App 的设计，梳理出 App 的业务流程，把每个业务流程列出。

（2）把每个业务流程可能会遇到的问题整理出来。

（3）根据整理出的问题，探讨可行的技术解决方案。

（4）把（3）中的所有技术解决方案有机融合，就是一个 App 后台的初步架构。

另外，从建立架构的流程可得知，架构设计有以下特点。

1. 架构是和业务紧密相关

每个 App 都有独自的业务逻辑，遇到的问题也不会一样，解决方案也不一样，因此架构也不尽相同。

笔者经常在 QQ 上被网友提问：“App 后台应该采用什么架构？”笔者不了解相关的业务逻辑，不知道会遇到哪些问题，不能帮助网友确定其所需的技术方案，架构也根本无从谈起。

2. 架构的演变是由业务驱动

当 App 处于不同的发展阶段，架构上也需要做变化。

例如，App 刚上线的时候为了快速开发，查询用户的数据这个功能是每次查询数据库，随着用户量的增大，数据库的查询压力也随之增大，可能就要考虑缓存，或者把数据的查询迁移到 NoSQL 数据库。

但同时要考虑一个问题，如果初期架构不合理，到了后期因为业务的发展需要改变架构是很困难的。但是初期要弄一个好的架构，又可能耽误后台的研发进度，使 App 的上架时间推迟，这时又要面对巨大的资金和时间压力，具体怎么取舍，需要认真考虑，马虎不得。

3. 架构不是为了炫耀技术

架构是为了满足业务的需求而设计的，技术人员不该过度设计，如学了一堆最新、最炫的技术并将其放进架构，而不是根据实际的需求来做。

技术是为了满足业务而存在的。过度设计不但延误了 App 的研发周期，也可能给运维带来很多不必要的麻烦。

总结：

在App成长的过程中，后台的架构也需要不断成长，技术人员也需要一起成长。

关于App后台架构的更多介绍，请参阅“第9章 App后台架构剖析”和“第10章 App后台架构的演进”。

1.3 App和App后台的通信

开发者经常问：App和后台的通信是用HTTP协议还是私有协议？是用长连接还是短连接？读者阅读本书后可以解除上面的疑问。

1. 用HTTP协议还是私有协议？

间谍电视剧经常能看到间谍们的书信是使用暗号的，就算书信被敌人截取，敌人也需要耗费一定的时间才能解开书信中的秘密。

在电影《阿凡达》中为提升故事真实性，导演詹姆斯·卡梅隆甚至找到语言学家Paul Frommer教授，创造了一种属于纳美人的语言。大家看这部电影的时候，根本听不懂潘多拉星球上的纳美外星人到底说什么。纳美语共有大约1000个单词，全球能够掌握其语法的人只有创造这门语言的Paul Frommer一人，而且就算他本人也仍然在学习如何更流畅地说纳美语。

如果间谍们使用的都是大家都懂的中文，敌人截取到书信，就能立刻知道里面的内容。如果潘多拉星球上的纳美外星人说的是中文，那么很多人一听就知道其具体内容。

App和App后台的通信也同样，可以分为使用通用语言通信和使用暗语通信两种方式。

通用语言有很多种，例如英语和中文。协议就相当于一套语言，双方都知道每个字节的具体含义。网络通信中有很多通用协议，其中HTTP协议是使用得最广泛的一种。大多数开发语言都支持通用协议，有大量的成熟模块供程序员调用，方便程序员解析这些通用协议的内容。

使用私有协议就相当于使用暗语通信，其类似于开发一套新的语言。私有协议对协议的封装和拆解工作量大，App程序员和后台程序员都要增加额外的工作量，而且私有协议对程序员的设计能力要求高，从Web网站转向移动开发的开发者上手有一定的困难。除非开发者对App的安全性和性能要求高，不然选择HTTP协议就足够。

2. App和App后台通信使用长连接还是短连接？

假设读者通过手机拨打另外一个人的手机，手机通话费用非常便宜（甚至可以忽略），但这个打电话过程有两个特点。

（1）一部手机同一时间只能接听一个电话。

（2）一部手机接听电话前非常麻烦，要拨号，要等接接听，这些过程需要耗费一段时间。

App 和服务器通信使用长连接还是短连接这个问题，可以使用上述的手机拨打电话的模型理解：是一直保持着通话，还是有需要时才拨号通话？

当 App 和服务器通信使用长连接，就相当于一直保持着通话，服务器能保持的通信数量有限，如果达到通信数量的限制，必须增加服务器才能让其他 App 继续和后台通信。这种通信方式，多数是使用 Socket 或 WebSocket 连接长时间连接，对程序员的素质要求高，开发困难，除了手游和聊天推送服务外，不建议使用。

当 App 和服务器通信使用短连接，就相当于需要时才拨号通话。这种通信方式主要是 HTTP 协议，是现在主流的通信方式，开发效率高，有大量的第三方软件可供开发人员使用，而且大多数开发人员对 HTTP 协议有一定的了解，能大大减少开发人员的认知成本，推荐使用这种方式。

3. App 和后端是怎么通信的？

相信读者都用过银行的柜员机（ATM）的查询余额、转账、取款等功能。

当用户在 ATM 取款时只需要输入取款的金额，隔一会儿钱就出来，如果因为有什么问题不能取款（例如超过取款金额的限制），屏幕上也会显示出错误的信息。

在整个过程中，用户只要输入金额就能获得结果（取出钱或屏幕提示不成功），至于柜员机内部是怎么处理，用户不需要知道。

用户使用 ATM 的流程如图 1-1 所示。

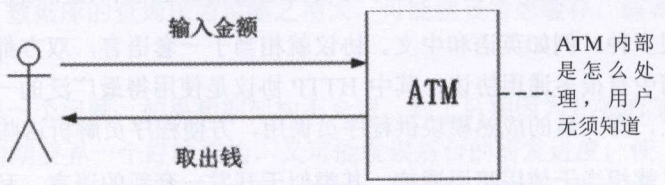


图 1-1　用户使用 ATM 的流程

ATM 这种把内部的处理遮蔽的做法极大方便了用户的使用。

同样在 App 后台也只提供了一系列的功能给 App 使用，这系列的功能以 API 的形式提供。

API 的定义：API（Application Programming Interface，应用程序编程接口）是一些预先定义的函数，目的是提供应用程序与开发人员基于某软件或硬件得以访问一组例程的能力，而又无须访问源码，或理解内部工作机制的细节。

当App调用后端提供API的时候，只需要明确下面3点：

（1）这个API的用途：在ATM的例子中，是取款，还是查询余额，还是转账？

（2）输入什么：在ATM的例子中，使用取款功能要输入金额。

（3）结果是什么：在ATM的例子中，取款是成功还是失败？

至于API内部是怎么处理，App无须知道。

4. 后端是返回给API的数据格式

API一般是以HTTP的形式调用的，通过HTTP传入参数返回数据。那么，App后台以什么样的格式返回数据呢？

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式。JSON采用完全独立于语言的文本格式，但是也使用了类似于C语言家族的习惯（包括C、C++、C#、Java、JavaScript、Perl、Python等），这些特性使JSON成为理想的数据交换语言。同时JSON易于阅读和编写，也易于机器解析和生成。下面是一个JSON格式的例子：

```
{
  "age": 11,
  "name": "jeff"
 }
```

另外一种常见的数据格式XML，其用来标记数据、定义数据类型，是一种允许用户对自己的标记语言进行定义的源语言。它非常适合万维网传输，提供统一的方法来描述和交换独立于应用程序或供应商的结构化数据。下面是一个XML格式的例子：

```
<?xml version="1.0" encoding="UTF-8"?>
<name>jeff<name>
<age>11<age>
```

读者比较一下上面举例的XML格式和JSON格式的数据，表示相同的数据内容，XML格式比JSON格式需要花费更多的字节。从上面的对比很容易看出，JSON格式更省流量，所以现在大多数API（例如新浪微博的开放API）都是以JSON作为返回数据的格式。

App和App后台的通信过程如图1-2所示。

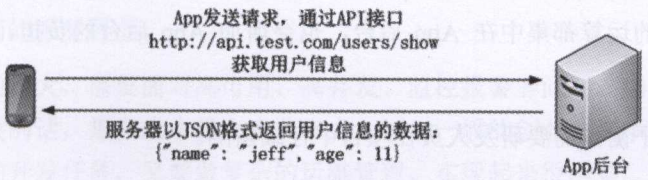


图1-2 App和App后台的通信过程

1.4 App 后台和 Web 后端的区别

很多从 Web 后端转到 App 后台的读者经常很茫然，不知道这两者之间有什么区别。本书通过例子，分析 Web 后端和 App 后台的区别，使读者能更好地把握 App 后台的架构。

1. App 后台要慎重考虑网络传输的流量，主要在 API 设计、图片处理上

现阶段手机上网的资费还是要按照流量算的，一般的 3G 用户，每个月的流量几百MB，4G用户，每个月的流量也只有几 GB。

如果不考虑网络传输的流量，一张图片就占了几百KB的空间，流量用得飞快。

在前面的文章“1.3 App 和 App 后台的通信”中提到，API 的返回结果一般是 JSON 格式，使用 JSON 格式的一个重要原因是，同样的内容，用 JSON 格式更省流量。

App 下载的图片也一样，一个节省流量的处理方法是让 App 下载经过压缩的图片（一般是几十 KB 以下），当用户需要查看原图时才下载原图。

2. 移动手机弱网络环境

移动手机因为不断移动的特性，特别是在高速移动的过程中，信号时有时无。

因此 App 后台发给 App 的信息是无法保证一定到达 App 的，极有可能的情况是：当 App 后台发送信息的时候 App 是连接网络的，但发送的过程中网络断开了，这样 App 就无法收到消息。

例如，推送系统中 App 要保存接收到的消息编号。服务端发送了编号为 1、2、3 这 3 条推送消息给 App，App 接收消息的过程中网络断开了，App 端只收到消息编号为 1、2 的消息，这意味着编号为 3 的消息丢失了，但是推送服务器是认为编号为 3 的消息已经推送成功了。

3. 手机电量有限

普通的手机电池被充满后能用一天左右，如果在 App 端做大量的网络请求和运算，手机的电量将消耗得很快。

但如果把所有的运算都集中在 App 后台，也会增加 App 后台的负担，严重的话会造成服务器宕机。

这两者之间的平衡，需要研发人员在项目中仔细斟酌。

1.5 选择服务器

对于很多刚入行的朋友来说，不清楚应该选择什么样的服务器提供商，是选择传统的 IDC，还是选择现在热门的云服务器呢？在本书中，笔者通过对比传统的 IDC 和云服务，简单阐述一下选择服务器的问题。

1. 是选择传统的 IDC 还是云服务器

App 产品经常会出现在毫无征兆的 App 访问量爆发的情况。如果出现了 App 访问量爆发的情况，解决访问的压力最快、最有效的方法是升级服务器的硬件，如升级 CPU，升级内存容量或者升级带宽。

传统的 IDC 要升级 CPU 或升级内存容量的流程如下。

- 和客户经理商谈所需硬件的价格或在线选择具体的配置。
- 在线支付或银行转账。
- 确认钱到账后，等待 IDC 安排工作人员升级硬件。

在这个流程中由于需要人工的介入，很难做到几分钟内完成升级硬件。

使用云服务器升级硬件就很简单，流程如下。

- 在用户后台选择升级后的硬件配置。
- 通过网络支付。
- 重启服务器，升级就完成了。如果只是升级带宽，甚至不用重启服务器。

整个过程算起来不用 5 分钟，简单、快捷、方便。

而且现在的云服务提供商除了提供服务器外，还提供下面这些服务。

- 负载均衡。
- 云数据库。
- 云内存存储。

App 上线初期，一般开发者都在一台服务器上搭建所有的服务，但随着 App 的发展，这些服务需要部署在不同的服务器上。

随着业务规模的增大，需要面对高可用、高并发、监控报警等问题。如果这些运维问题都要研发人员自行解决的话，那在成本投入上非常大，因为一般的创业公司中研发人员就一两个人，既要保证平时的开发任务，又要做复杂的运维管理，实现起来很困难。而且研发人员也不是全能的，对于没用过的专业运维知识也需要一定的学习成本，就算研发人员学会了，真正部

署实施效果怎么样还很难保证。在这些情况下就能体会到云服务的优点，由云服务器的提供商来负责运维，高可用、高并发、监控报警等方面都能靠云服务器提供商保障，企业使用云服务就能大大减轻运维方面的压力和研发的成本。

2. 笔者建议

笔者在网络上经常被问：需要选择什么样的服务器配置？这个问题笔者没法回答，因为需要综合考虑用户量、业务逻辑等因素。不过笔者建议项目初期的硬件配置可以稍为逊色点，随时监控主机资源的状态，当预估当前的配置不能应付业务上的需求时就考虑升级硬件，毕竟现在升级或者购买云服务器都非常方便。

1.6 选择编程语言

在 QQ 上，经常看到创业团队的创始人一直都招不到技术人员，了解后发现其中一个原因是其所要求掌握的编程语言偏门。读者通过阅读本节，可以详细了解选择开发语言的核心原则，使各位读者对选择哪种开发语言更加有把握。

选择编程语言笔者就一个核心原则：**选择符合业务场景的最热门的编程语言**。

1. 每种编程语言都有自己擅长的业务场景和性能特性

例如开发聊天服务器，选择了 PHP 开发那真的太不合适，PHP 这种语言怎么能适合聊天服务？

假如选择开发 Web 网站的编程语言，PHP 比 Golang 合适多了。

2. 选择开发效率最高的编程语言

很多编程语言适用的业务场景是重叠的，性能上的差距在项目初期也不明显，那么这种情况下应该怎么选择？

笔者的答案是，选择开发效率最高的编程语言。开发效率高意味着能快速推进产品的研发，有利于产品的迭代，大大减少资金和时间成本，在项目初期，确保产品能活下去是开发人员考虑的第一要点。

3. 一个大忌是用两套不同的编程语言维护一个相同的业务逻辑

曾经有个创业者咨询笔者，其项目有 App 后台和网站两部分，其想用 Java 来开发 App 后台，用 PHP 来开发网站后台，两部分有大量的业务逻辑是相同的。笔者听完这个创业者的说明后立刻表明：这种情况就是重复造轮子。用两套语言去维护相同的业务逻辑，在开发效率上有提升吗？如果需要修改重叠部分的业务逻辑，那要分别使用两种编程语言去修改，这不是浪

费大量的开发时间吗？

4. 一个系统中，不同的业务逻辑可以用不同的编程语言实现

例如，Web网站部分可以使用PHP，推送服务器部分可以使用Golang或者Erlang实现等。

最后，如果真的没法决定采用哪种编程语言，还有一个办法，找业务逻辑差不多的同类产品，看其招聘要求参考一下。

1.7 快速入门新技术

App后台的工作经常会接触到新的技术，作为一名后端人员，面对开发的压力，快速地入门新技术并把其融入到项目中，这已经成了一名后端人员的必备技能。在本节内容中，根据笔者总结出来的一个核心思维模式，介绍4种快速入门新技术的方法。

1.7.1 思维模式

App后台技术是十分复杂和多种多样的，开发者要快速入门，必须要有清晰的思维模式帮助我们拨开云雾、探知问题的根本，不然就很容易在纷繁复杂的技术中迷失。

曾经有名开发者在QQ上问笔者：“Openfire（Openfire是一个开源的聊天服务器）的群聊是怎么实现的？”笔者在和开发者沟通的过程中发现其连Openfire都没运行过，更别说用过Openfire的管理后台，连Openfire最基本的功能都没用过的开发者，又怎么可能明白Openfire的群聊功能呢？

在认识一个新的事物时，一个特点就是“从整体到局部”。如果只是“只见树木不见森林”，就容易迷失在无数的细节当中。

笔者认为最重要的是“抓核心、做减法”。从纷繁复杂的万物万象中，发现重点是哪里、关键是什么，抓住这个核心，就能做到化繁为简。

对于软件技术来说，核心问题就2个。

- 软件的适用场景。
- 软件的运行原理。

本书的知识点就是围绕着上面2个核心问题展开的。

1.7.2 4种快速入门新技术的方法

下面介绍4种快速入门新技术的方法。

1. 阅读软件安装的 README 文件和 INSTALL 文件

很多软件的安装包中都有个“README”文件，顾名思义，其名字已经提示让开发者去阅读，这个文件是非常重要的。这个文件中有关于这款软件的详细的功能说明。

INSTALL 文件是关于这款软件最简单的安装方法，里面描述了这款软件是怎么运行起来的。

但很多时候 README 文件和 INSTALL 文件会合并成一个文件，例如“Redis”这款软件中就只有 README 文件。

2. 阅读官网的文档

有些开源软件的官网文档中会有一个栏目叫“how to start”或“quick start”，类似这些名称，里面的内容是教开发人员怎么快速部署安装软件，实现这款软件的基本功能，运行这款软件。开发人员通过阅读这些栏目，可以对这款软件有基本的了解。

3. 阅读源码里的 example 文件夹

某些开源软件的开发者会在源码中附上代码例子，放在 example 文件夹、test 文件夹，或其他文件夹里。开发人员仔细阅读这些代码也能快速入门新的技术。

4. 在搜索引擎网站中搜索

如果按照上面的 3 种方法，还是不得头绪，那就只能搜索别人写的相关入门教程。

例如需要搜索 Redis 的入门教程，那就使用关键字“Redis 入门教程”或“Redis 教程”在搜索引擎网站中搜索相关的教程。在互联网这个知识爆炸的时代，确实涌现了很多优秀的教程，能节省很多时间。

1.8 App 是怎样炼成的

很多刚进入 App 后台这个行业的读者，有的是之前没有接触过这个行业，有的是只在学校学习了基本的技术知识，不知道开发 App 产品的流程是怎么样的，因此心里会有一股恐惧，听着别人口中的一大串相关术语，也不知道怎么回事，更谈不上和别人交流。在本书中，根据本人在创业公司的经历帮助读者了解 App 开发的基本流程，助其迈入 App 开发的大门。

1.8.1 项目启动阶段

在一个 App 项目启动之前，由产品经理（在创业公司里， 产品经理一般都是公司的创始

人）介绍个人对 App 的想法，例如 App 是做什么业务，有哪些界面，每个界面上有哪些按钮，每个界面之间是怎么跳转等。产品经理根据以上的这些想法画出原型图。

原型图一般是用 Axure 这款软件制作的。产品经理用 Axure 制作完原型图后把原型图导出为一堆 HTML 文件，在浏览器中打开 index.html 文件就能看到原型图。

原型图的例子如图 1-3 所示。

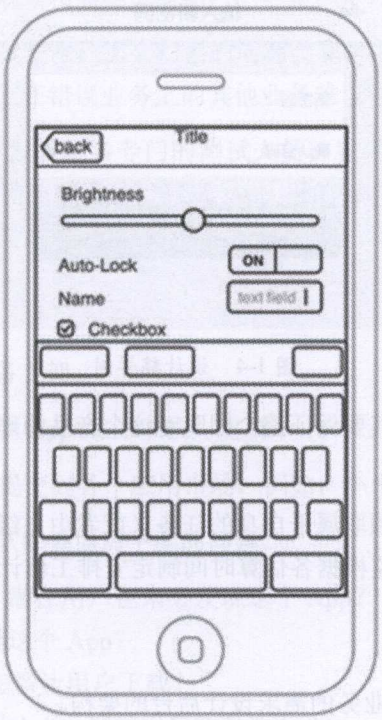


图 1-3　原型图例子

从上面的原型图中可看出，这个原型图只是简单描述了 App 的基本界面，界面极其简陋。开发人员依靠这份原型图是很难开发 App 的（如果完成了原型图就着手开发也行，只是研发人员在估算开发界面所需的工作时间上会非常不准确）。

原型图出来后，产品经理就把原型图交给 UI 设计师出设计稿。UI 设计师开始根据自己的美术功底美化原型图：给里面的每个元素都配上合适的颜色；调整整个界面的布局，按钮的大小、位置、颜色等，务必使整个界面看起来更美观；标示里面的文字的坐标、使用的字体等。

设计稿的样例如图 1-4 所示。

甚至有的 UI 设计师还要负责设计交互，例如，A 界面是怎么跳到 B 界面，是从上往下跳

转，还是从下往上跳转。

当 UI 设计师出了设计稿后，产品经理和整个项目相关的人员开产品会议。产品会议里，产品经理亲自介绍原型图，把整个产品的业务逻辑用原型图向相关的人员演示，还有回复大家对产品的疑问。

图 1-4　设计稿样例

这个产品会议非常重要：其要保证整个团队对这个产品的理解一致，从而确保接下来开发工作顺利进行。

产品会议后，相关的人员领取属于自身的任务（或者由上级分派），相关的人员同时估算研发时间，研发总监或技术总监根据各估算时间制定安排工作计划。

1.8.2　研发阶段

后台开发人员根据产品和业务的需求设计后台的架构。

Android 研发人员和 iOS 研发人员可以先设计前端的架构，或者根据设计稿开始先做界面，凡是需要和后台交互的部分先不做。

当后端的架构设计完成后，后端开发人员通过下面 3 点先初步设定 API 接口。

1. API 是有什么用的？
2. API 的输入参数是什么？
3. API 返回什么数据？

后台研发人员对 Android 研发人员和 iOS 研发人员说明其设定的 API 接口，让其了解相关内容。这些 API 接口初期先不用实现其功能，只需要返回一些测试的数据以便前端人员开发，后台研发人员在研发的过程中慢慢把这些接口的功能实现。这样前端和后端的开发进度都不会

耽误。

当然，这些 API 接口以后不排除有改动的可能，为了保证信息的通畅，任何对 API 接口的改动请及时通知相关人员。

1.8.3 测试阶段

当制订开发计划时就应该规划功能测试周期，一般是一个月测试一次为宜。

如果开发了两三个月才测试一次，那么积累的问题会非常多，如果对某个业务理解错误，那么过长的研发周期也会使建立在错误业务上的其他业务难以修复。

创业团队中大多数情况下没有配备专门的测试人员，更多的情况是“人人都是测试员”。整个创业团队里的每个人都充当测试人员，测试 App 里的每个功能，记录下所发现的问题，整理后提交给相关的负责人修复。

1.8.4 正式推出阶段

App 测试完毕后就开始准备上架。

如果是 iOS 应用，就提交到 App store 审核，需要 7~15 个工作日。

如果是 Android 应用，就提交到各个应用市场，例如，小米应用市场、豌豆荚、应用宝等。

在正式推出阶段，每个团队都要面对下面的问题。

- 提交到各个应用市场，潜在用户也未必发现这个 App？
- 怎么让更多的用户了解这个 App？
- 用户了解这个 App 后怎么让用户下载？
- 怎么保证用户经常使用 App，增加用户的黏性？

这些问题是创业团队中市场推广人员和运营人员需要解决的，当然团队中的成员也可以为解决这些问题贡献一份力量。

总结

以上是整个 App 研发流程的概括说明。

上面的项目管理部分描述得比较简单，详细的项目管理内容将在下面的章节“1.9 最适合 App 的开发模式——敏捷开发”中介绍。

创业团队中的职权很难分清，例如在创业团队中，App 后台人员都要兼职做运维，甚至是前端，理由很现实：没额外的资金聘请人员负责这些工作，但这些工作总要做，最后就只能由

App 后台人员兼任。

而且很多团队中经常有身兼多职的情况。例如笔者认识的一个创业团队只有 4 人：一个创始人，1 个 App 后台人员，1 个 Android 研发人员，1 个 iOS 研发人员，按照这样的人员分配，研发外的所有的工作都由创始人负责。

在创业团队中工作能力成长快，大量的问题都要开发人员处理，同时在创业团队里对人员技能要求比较高，要一专多长，在有需要的时候加班也不是什么稀奇的事情。

1.9 最适合 App 的开发模式——敏捷开发

传统的软件开发模式需要经历问题评估、计划解决方案、设计系统架构、开发代码、测试、部署和使用系统、维护解决方案这个过程，如图 1-5 所示。

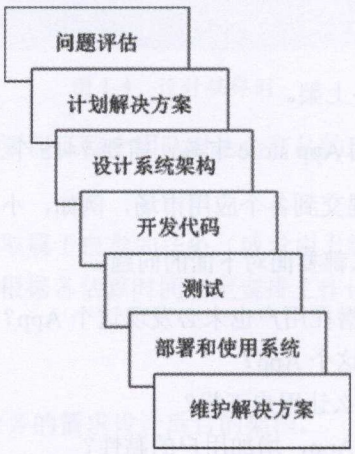


图 1-5　传统软件开发过程

采用传统软件开发模式的最大问题是开发周期过长，迭代速度慢。移动互联网行业发展速度快，需求不断变化，产品更新迭代的频率高，基于移动互联网的以上特点，笔者曾经的团队在开发产品的过程中放弃了传统的瀑布流开发模型，引入了 Scrum 这个敏捷开发框架，下面谈谈当时笔者所在团队实施敏捷开发过程中的一些经验。

Scrum 简介：Scrum 是一个敏捷开发框架，是一个增量的、迭代的开发过程。在这个框架中，整个开发周期包括若干个小的跌代周期，每个小的迭代周期称为一个 Sprint，每个 Sprint 的建议长度为 2~4 周。在 Scrum 中，使用产品 Backlog 来管理产品或项目的需求，产品 Backlog 是一个按照商业价值排序的需求列表，列表条目的体现形式通常为用户故事。Scrum

的开发团队总是先开发对客户具有较高价值的需求。在每个迭代过程中开发团队从产品 Backlog 挑选最有价值的需求进行开发。Sprint 中挑选的需求经过 Sprint 计划会议上的分析、讨论和估算得到一个 Sprint 的任务列表，称为 Sprint Backlog。迭代结束时团队将交付潜在可交付的产品。

当时笔者团队中的 App 项目的 Sprint Backlog 大概为 4 周时间，其中包括 2 天的 Sprint 计划会议时间，2.5 周的开发时间，开发的日子里每日例会必不可少，1 周的测试修复 Bug 时间，1 天的 Sprint 评审会议和 Sprint 回顾会议。

Scrum 流程如图 1-6 所示。

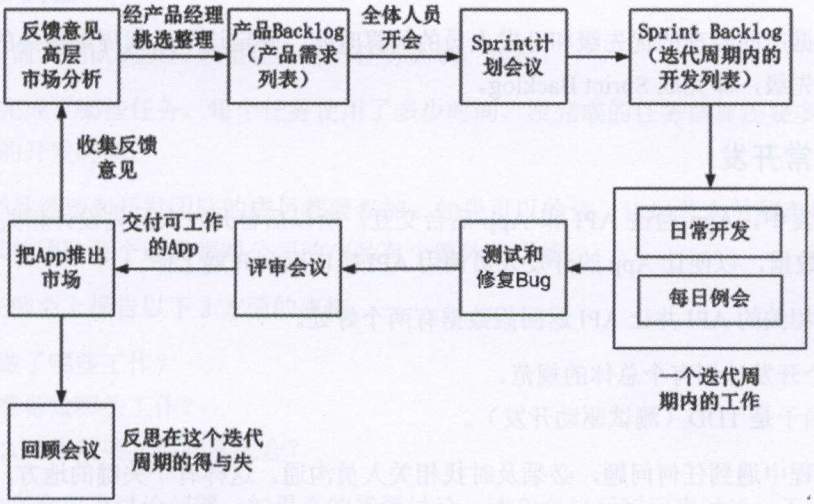


图 1-6　Scrum 流程

1.9.1　Sprint 计划会议

Sprint 计划会议前，产品经理所要实现的产品需求（产品 Backlog）以用户故事（即从用户的角度去描述用户所需的功能）的形式确定下来，并画出原型图，UI 根据原型图完成设计稿（在 Sprint 计划会议前出设计稿很重要，因为设计稿对估算时间的影响非常大）。产品经理同时确定各个产品需求的优先级。

Sprint 计划会议期间（一般是 2 天），开发团队的成员不应该做任何开发工作，要将全部精力放在把产品需求分解成一个个开发任务，并估算开发时间。

估算开发时间需要注意以下几点。

1. 对于所需要使用的新技术，要估算学习和调研的时间。

2. 根据统计，每个程序员每天的有效工作时间是 5 个小时左右，其他时间都被沟通、喝水、休息、上洗手间等琐事占据，如果某个任务估算超过 5 个小时，那就代表了这个任务完成需要超过一天的时间。

3. 开发人员对于开发任务的估算尽可能精细，一般来说，每个任务的估算时间不应该超过 5 个小时，如果超过 5 个小时，就应该把这个任务再细分为多个更小的任务。只要尽可能精细地估算任务，总体估算时间是大概精确的，因为有的任务估算的时间比实际完成的时间多，有的时间任务估算的时间比实际完成的时间少，平衡后总量是差不多的。当然有时会遇到意料之外的技术难题，这时所估算时间就要大打折扣。

最后根据产品经理的优先级和开发人员的估算时间，确定这个迭代周期最终的开发任务和其对应的优先级，即完成 Sprint Backlog。

1.9.2 日常开发

App 开发中，App 通过 API 和 App 后台交互，所以后台开发人员先设计相关的 API 并让 API 返回假数据，以便让 App 的开发人员调用 API 接口顺利开展工作。

先设计相关的 API 并让 API 返回假数据有两个好处。

- 整个开发计划有个总体的规范。
- 相当于是 TDD（测试驱动开发）。

开发过程中遇到任何问题，必须及时找相关人员沟通。这样有个关键的地方，沟通可能会打断别人工作，打断别人的工作有两种情况：一种是别人正在投入、忘我地进行某项关键性工作；另外一种是别人正进行一些零碎的、不重要的工作。对于第一种情况，打断别人的工作后别人心情也不好，而且其重新切换到投入、忘我的工作状态很困难；对于第二种情况则没多大关系。为了保证沟通的效果，可以采用下面的方法。

- 如果不是非常紧急的问题，可以等相关人员休息的时候再沟通。
- 解决一个问题，先梳理情绪，再梳理人际关系，最后才是问题本身。多微笑，别苦着脸，平时待人和善，说好话，存好心，做好事，沟通的时候对事不对人。
- 对于关键性工作可以试一下番茄工作法：设计一个很短的时间周期（例如 25 分钟）专注于工作，中途不允许被任何人打扰，过了这个时间周期后休息 5 分钟。

Scrum 中有个关键的职位“Scrum master”，在创业开发团队中，Scrum master 一般是由技术总监担任，团队和外部的沟通必须统一通过 Scrum master。例如市场部、运营部的同事有什么需求要开发团队完成，必须要经 Scrum master 同意后再由 Scrum master 和开发团队沟通。

如果开发人员有重大的决策，也必须经 Scrum master 同意。Scrum master 的最大作用是屏蔽外部对开发团队的影响，使开发的进度和开发的效率得到保证。

团队建设采用定期团体活动，当时的团体活动是周四下午集体去打羽毛球。团体活动不但能加强团队的凝聚力，而且运动后身体沉闷的感觉都消失了，变得神清气爽，活力十足，开发效率倍增。

在开发过程中需要注意：一个 Sprint Backlog 中，需求不能变更，UI 确定后原则上只能做小修改（但这点无法得到保障）。产品有新的需求，下一个 Sprint Backlog 再考虑。

1.9.3 每日例会

每日例会前，团队成员应该整理各自的任务列表，包括：

- 昨天完成了哪些任务，每个任务使用了多少时间，没完成的任务估算还要多少时间。
- 剩余的开发时间。

例会中产品经理和开发团队的成员都要参加，如果可以的话，让运营人员和市场人员也参加，这样可以使团队每个成员都对公司的产品有个整体的了解。

每个人在例会上报告以下 3 方面的事情。

- 昨天做了哪些工作？
- 今天准备做哪些工作？
- 有什么工作需要其他同事配合？

注意避免在会议上讨论问题，如果真的需要讨论，请在会议后和同事讨论，不要浪费整个团队的时间。

1.9.4 测试和修复 Bug

产品开发完成就进入测试和修复 Bug 的阶段。如果人手不足，可以使用交叉测试的方式，即 Android 开发人员测试 iPhone 的 App，iPhone 开发人员测试 Android 的 App，后台、运营、UI 等人员看情况分配测试任务。

测试人员把测试得到的问题提交到 Bug 管理软件，每个 Bug 应该包含 3 部分。

- 问题描述和重现步骤。
- 测试人员。
- 负责解决这个问题的人员，如果测试人员不知道具体负责人，把问题提交给技术总监，由技术总监指定解决问题的研发人员。

1.9.5 评审会议

在测试和修复 Bug 完后全体人员开评审会议。

相关的开发人员在评审会议中向全体人员演示 App 的功能。iPhone 的演示可以使用一个收费的工具，该工具把 iPhone 屏幕的影像传输到 Mac 电脑，再通过 Mac 电脑传输到投影仪，Android 上没找到哪个好用的演示工具（后来网友提示 360 手机助手可以实现演示功能）！当时我们的方法是 Android 演示的时候，用 iPhone 的摄像头对着 Android 机，通过 iPhone 的收费工具在投影仪上观看。

1.9.6 回顾会议

研发完成后开回顾会议，每个成员都在会议中提两点。

- 这轮迭代过程中做得好的地方。
- 这轮迭代过程中做得不好的地方。

这个过程走两轮，即每个成员都要提两点做得好的地方和两点做得不好的地方。

注意当一个成员提出自己的意见时，其他成员不做任何的评价。

1.9.7 及时反馈

精益理念中很重要的两点是快速反馈和快速迭代。快速迭代是通过 Scrum 这个敏捷开发框架实现的，但快速反馈呢？产品投入到市场后，怎么快速收集用户的反馈呢？

当时项目采用的方法如下。

- 建立相关的 QQ 群，收集意见。
- 在 App 中，有个意见反馈的功能，能把反馈的意见发送到服务器。
- 后台中有个系统的账号。每个用户注册后就自动加这个系统账号为好友，可以随时通过聊天功能向这个系统账号提问题。产品经理经常登录这个系统账号和用户交流。

1.9.8 总结

敏捷开发不是万能药（世界上也不会有万能药），团队如果不结合项目的实际情况，觉得敏捷开发时髦就生搬硬套地把敏捷开发引入到项目中，那是得不偿失的。敏捷开发更适用于需求多变、开发周期短的项目，例如 App 的开发，对于大型的航天、银行、证券等项目，敏捷开发不一定合适。

第 2 章
App 后台基础技术

这章详细描述了 App 后台工作中会涉及的基础技术以及原理。通过本章的学习，读者能够了解一般的 Web 后台和 App 后台架构上各个组件的作用。

2.1 从 App 业务逻辑中提炼 API 接口

在 App 后台工作中设计 API 是很考验设计能力的。在项目的初始阶段，我们只知道具体的业务逻辑，那怎么把业务逻辑抽象和提炼从而设计 API 呢？读者通过阅读本书，可解答以上疑惑。

本节用笔者以前开发的 App“移客 ekeo”第 1 版的（以后的业务逻辑改了很多）业务逻辑为例。移客 ekeo 是一款以熟人社交和真实聚会为核心的社交工具，其以解决聚会难题为核心，用户通过移客 ekeo 快速发起聚会或者参与聚会活动，并能掌握参加者是否已经出发或者到达聚会地点。

本书是笔者根据刚入行时技术总监 Howard 的教导整理而成的，很感谢 Howard 的教导，笔者从中获益良多。

从业务逻辑到最终提炼 API 可分为下面 6 个阶段。

- 业务逻辑思维导图。
- 功能—业务逻辑思维导图。
- 基本功能模块关系。

- 功能模块接口 UML（设计出 API）。
- 在设计稿标注 API。
- 编写 API 文档。

2.1.1 业务逻辑思维导图

整个流程的第一步就是抽象出业务逻辑。

抽象就是把相同的东西先放在一起，这是抽象的第一步。业务流程就像一条河一样，从头走到脚，里面会有很多一样的东西，开发人员需要把一样的东西先抽象出来，就好比一栋楼的结构，有很多三角形、圆形、正方形，开发人员先要把这些东西抽象出来。

如果连抽象都搞错了，写接口也没什么用，只是在乱做而已。

所以说抽象的第一步就是，开发人员通过思维导图的形式把相同的业务流程抽象出来形成一个图表，然后通过关系箭头表现出来。

1. 首先是要用思维导图把结构关系列出来，包括里面的功能。
2. 把相同的元素整理出来，比如说，都是相同的推送、评论、图片上传，然后用相同的颜色，在 1.的图上面表示出来。

这样至少清楚哪些业务逻辑是一样的，如果连业务逻辑都没搞清楚，怎么能确保后面做的接口是完整的呢？把这一步搞清楚就知道了一个模块需要有哪些接口，那个时候再用 UML 图把接口和接口关系画出来，那个时候是抽象的第二步。

根据业务逻辑整理出如下的思维导图，如图 2-1 所示，相同颜色的部分就是相同的业务逻辑。

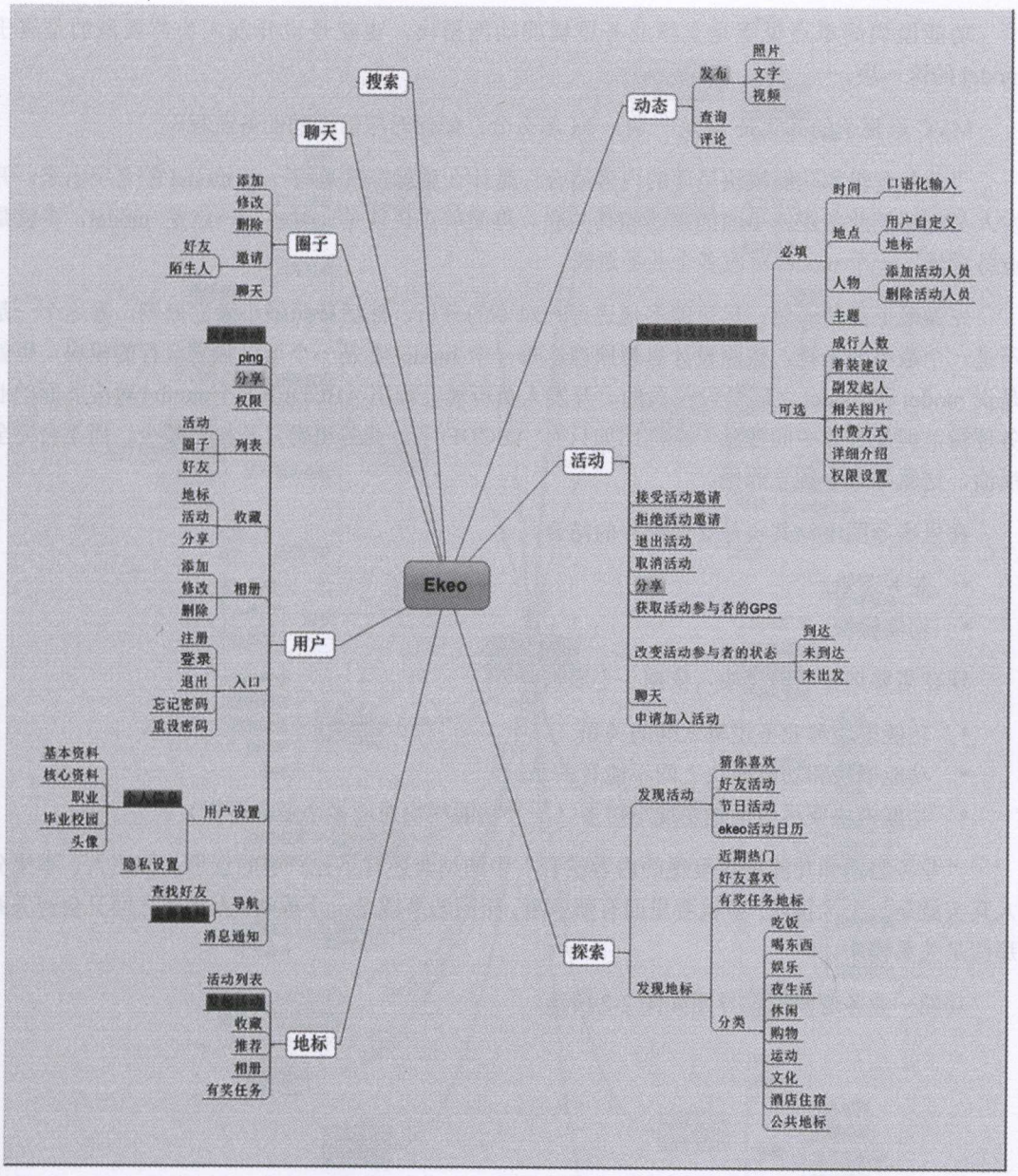


图 2-1　业务逻辑的思维导图

2.1.2　功能—业务逻辑思维导图

这步的核心是“业务逻辑和功能模块呈现的内容结合”。

功能模块简单点说就是支撑业务逻辑的功能模块，也就是说开发人员需要做的是属于 model 的这一块。

MVC 最复杂的其实是 M 这一块，M 怎么分，如何对应前面的业务流程？

“业务逻辑和功能模块呈现的内容结合”是什么意思？就是写一个 model 的名字出来，开发人员能够把业务逻辑里面的东西和其关联。再简单点说就是一对多，一就是 model，多就是业务逻辑，一个 model 对应多个业务逻辑。

先做最上面的一层，尽可能多地进行一对多的分析，然后按照最小独立原则，看这个一是不是一个最小的系统，按照设计思想原理：每一个 model 都是一个可以独立运行的模块，也就是说 model 和 model 之间是没关系的。开发人员可能会画出 ABCDEF 6 个 model 对应前面的业务逻辑，但是去掉中间任意一个，比如只有 ABCDF 了，业务逻辑只是相对减少，而不会完全崩溃，这就是最小独立原则。

在思维导图中，其实是 2 个部分的结合。

- 业务逻辑。
- 功能模块。

现在需要划分功能模块，依据 3 个原则。

- 功能模块和业务逻辑之间的关系。
- 功能模块和功能模块之间不能有关系。
- 功能模块要尽可能地实现一对多（一个功能模块对应多个业务逻辑）。

“业务逻辑和功能模块呈现的内容结合”中的结合还有另外一层的意思：按照人、事来分，人其实就是一个大模块，事就看里面有哪些事，相同的事就是一个模块，人和事之间又会有关系，那些是关系模块。

功能—业务逻辑思维导图如图 2-2 所示。

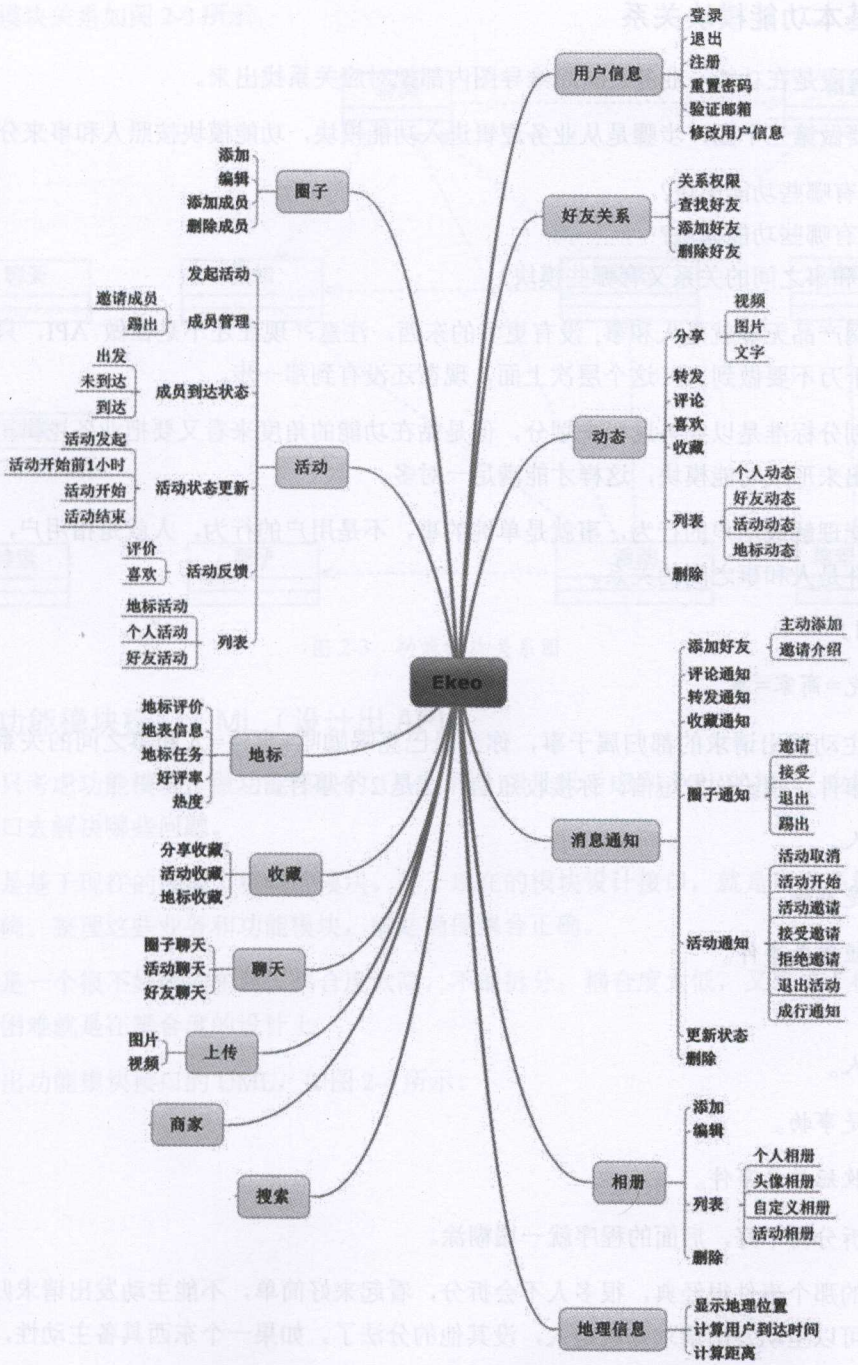


图 2-2　功能—业务逻辑思维导图

2.1.3 基本功能模块关系

这个阶段是在功能—业务逻辑思维导图内部把对应关系找出来。

现在要做第三个图，步骤是从业务逻辑进入功能模块，功能模块按照人和事来分。

- 人有哪些功能模块?
- 事有哪些功能模块?
- 人和事之间的关系又有哪些模块?

互联网产品无非就是人和事, 没有更多的东西。注意：现在还不是在做 API，只是在做功能模块，千万不要做到 API 这个层次上面，现在还没有到那一步。

事的划分标准是以业务逻辑来划分，但是站在功能的角度来看又要把业务逻辑里面相同的功能抽象出来形成功能模块，这样才能满足一对多。

事不能理解成用户的行为，事就是单纯的事，不是用户的行为，人就是指用户，事就是指事物，事件是人和事之间的关系。

你=用户=人

星巴克=商家=事

不能主动发出请求的都归属于事，你去星巴克喝咖啡=事件=人和事之间的关系。事是事物，不是事件，我给你发短信，你接收短信，这是 2 个事件。

我是人。

短信是事物。

我发短信是事件。

你是人。

短信是事物。

你接收短信是事件。

这个拆分得不好，后面的程序就一塌糊涂。

短信的那个事件很经典，很多人不会拆分，看起来好简单，不能主动发出请求归属于事。如果商家可以主动发出请求那就是人，没其他的分法了。如果一个东西具备主动性，其就是人，就是用户。

功能模块关系如图 2-3 所示。

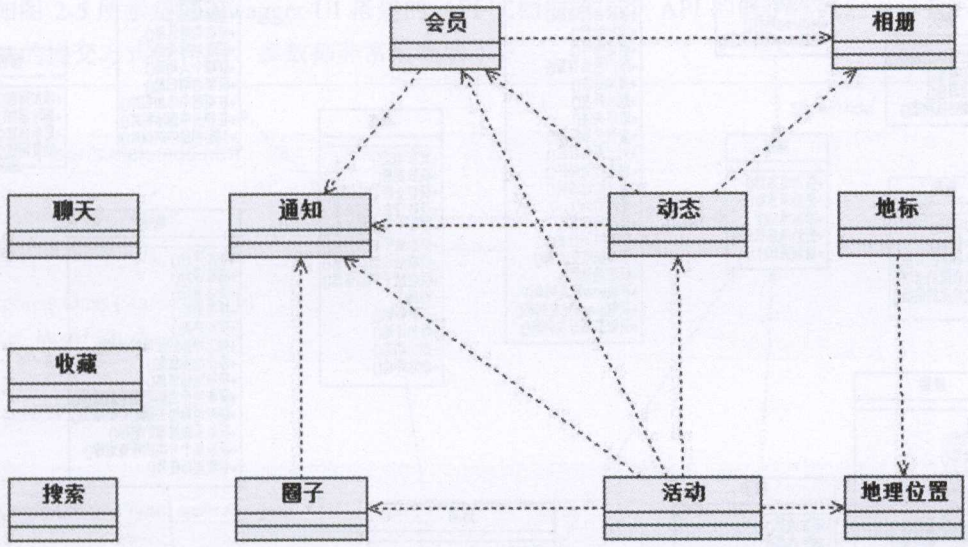


图 2-3　功能模块关系图

2.1.4　功能模块接口 UML（设计出 API）

现在只考虑功能模块，做功能模块的 UML 图，但是只考虑到 API 的接口，也就是说要设计哪些接口去解决哪些问题。

接口是基于现在的模块、粗略的模块。基于现在的模块设计接口，就是要提高耦合度，确保耦合正确。整理这些业务和功能模块，就是确保耦合正确。

耦合是一个很不好做的东西。耦合度太高，不能拆分；耦合度太低，又失去了模块化的意义。主要困难就是在耦合度的设计上。

整理出功能模块接口的 UML，如图 2-4 所示。

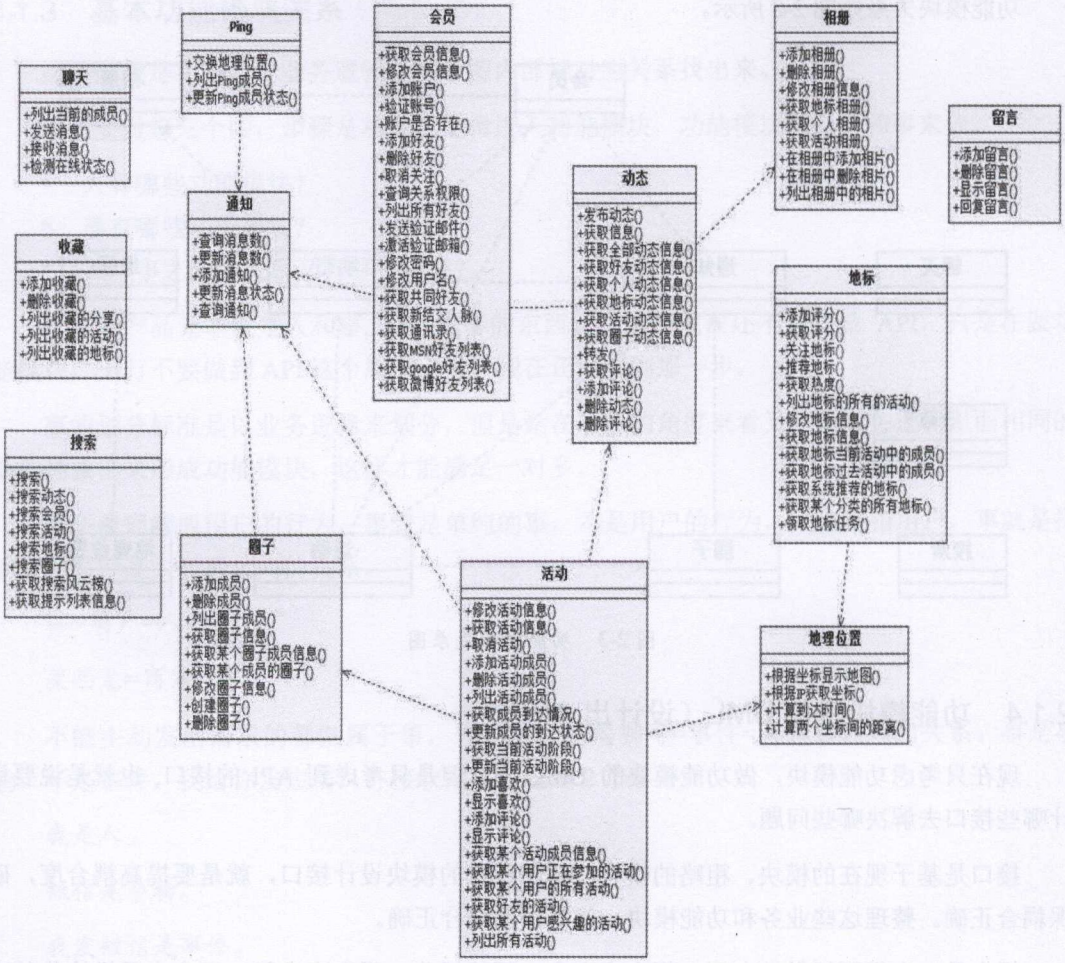


图 2-4　功能模块 UML 图

上面的图就是所整理的 API，把这些中文变成英文，就是熟悉的 API。

2.1.5　编写在线 API 测试文档

App 所需的 API 在线测试文档，既是一份在线 API 文档，也是一个在线测试工具，给开发沟通和测试带来极大的便利。当客户端程序员觉得某个 API 有什么问题，就会根据这个在线工具的执行结果和 App 后台人员讨论沟通。

API 在线测试文档使用 Swagger-UI 搭建。Swagger-UI 简单而一目了然，其基于 HTML+JavaScript 实现，只要稍微整合一下便能成为方便的 API 在线测试工具。项目的设计架

构中一直提倡使用 TDD（测试驱动）原则来开发，Swagger-UI 在这方面提供了很大帮助。

如图 2-5 所示是用 Swagger-UI 搭建的 API 文档中的一个 API 的例子，我们可以看到，整个 API 的提交方式、作用、参数都非常清晰明了。

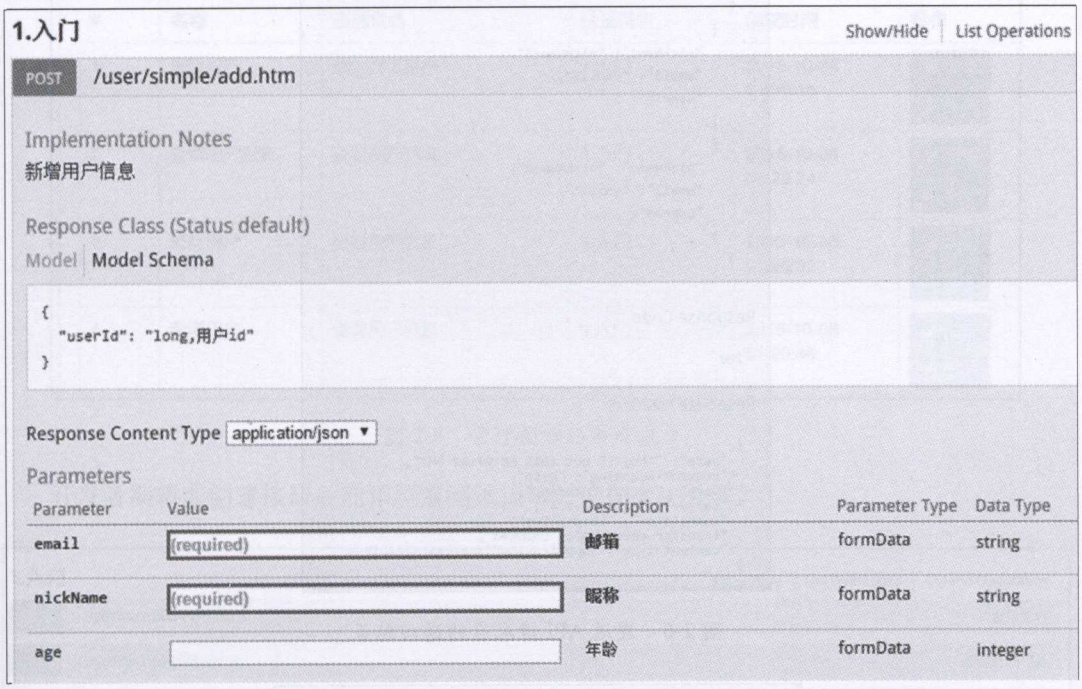


图 2-5　Swagger-UI 搭建的 API 文档例子

在 Swagger-UI 上发送一个 API 请求后，返回的结果也一目了然，如图 2-6 所示。

Swagger-UI 的部署需要和相应的 MVC 框架结合，而且在 Swagger-UI 定义的接口在 JSON 配置文件中读取，JSON 里面的配置项多，编写起来麻烦，一不小心就会出错，排查困难。笔者推荐一个网站：API 接口管理的网站（www.sosoapi.com）提供了在线编辑和预览测试接口的功能，其基于 Swagger-UI，方便地解决了 Swagger-UI 部署和编写接口的不便之处。

API 接口管理的网站提供填写表单的方式来创建新的接口，如图 2-7 所示。

Request URL

```
http://www.sosoapi.com/demo/user/simple/list.htm
```

Response Body

```
[
  {
    "nickName": "nickName1",
    "email": "email1",
    "userId": 1
  },
  {
    "nickName": "nickName2",
    "email": "email2",
    "userId": 2
  }
]
```

Response Code

```
200
```

Response Headers

```
{
  "date": "Thu, 15 Oct 2015 00:07:02 GMT",
  "content-encoding": "gzip",
  "vary": "Accept-Encoding",
  "server": "nginx/1.7.4",
  "connection": "keep-alive",
  "transfer-encoding": "chunked",
  "content-type": "application/json;charset=UTF-8"
}
```

图 2-6　发送 API 请求后的返回结果

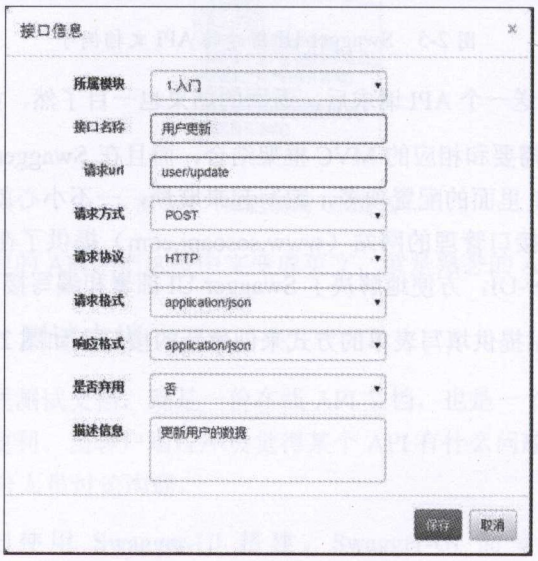


图 2-7　通过表单创建接口

在 www.sosoapi.com 中可以管理创建的所有接口，如图 2-8 所示。

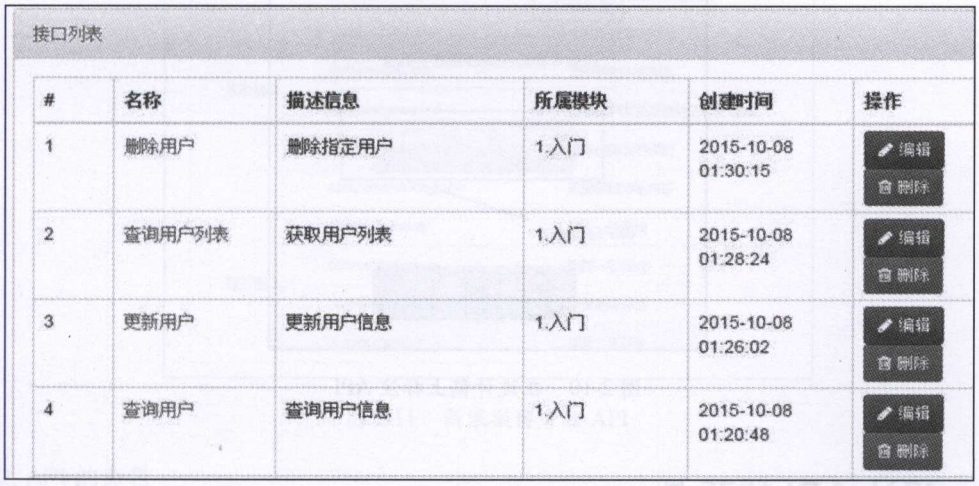


图 2-8　管理创建的所有接口

开发者编辑或创建接口后能预览或测试接口，如图 2-9 所示。

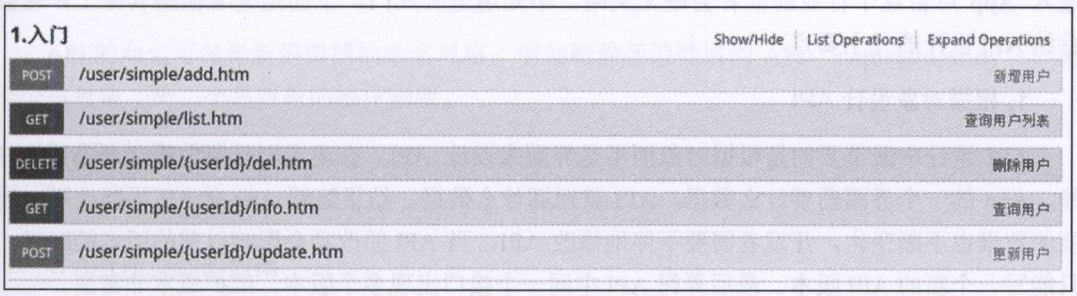


图 2-9　预览效果

如果开发者不想在 www.sosoapi.com 中测试 API，开发者编辑完毕后通过其提供的项目管理工具下载生成的 JSON 文件，导入到自身项目中的 Swagger-UI 即可。

2.1.6　设计稿标注 API

API 整理出来后，对于前端的开发人员来说还有一个大的障碍：不知道 App 在哪个界面调用哪个 API？为了方便前端人员，App 后台开发人员需要在设计稿上标注出哪个界面调用哪个 API，如图 2-10 所示。

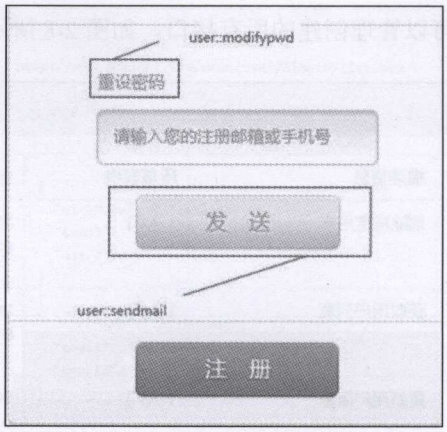


图 2-10　在设计稿上标注 API

2.2　设计 API 的要点

App 和 App 后台的交互是通过 App 后台提供的 API 实现的。API 的设计要点估计很多刚进入 App 后台这个行业的读者会毫无头绪，不知道怎么入门。下面是笔者根据实际工作经验总结 API 设计的几个要点。

1. 根据对象设计 API

API 设计中最重要的是根据对象而不是界面来设计 API。在笔者刚开始做第一个项目的时候，App 的一个界面需要什么数据，API 就返回什么数据。结果随着 App 的 UI 不断改版，需要的数据也不断变化，开发者需要不停地修改 API，当 API 的改动会影响以前的版本的时候，只能写一个新的 API 版本，最后弄得 API 中同一个接口出现多个版本，维护成本非常高。

后来笔者在 App 后台的重构过程中就根据对象设计 API。但根据对象设计 API 又有一个问题：一个大对象可能包含很多小对象，是一个 API 返回全部小对象，还是分为多个 API 返回呢？这点需要读者们根据自身的业务、技术、带宽等因素仔细考虑。

如果读者不明白什么是根据对象设计 API，可参考下面新浪微博开放平台的 API 的截图，如图 2-11 所示。

图 2-11 的微博开放 API 是评论相关的 API，对象是 comments（评论），后面紧接的是对象的动作或状态。例如 comments/destroy，删除评论（删除 comments 对象）。

评论		
读取接口	comments/show	获取某条微博的评论列表
	comments/by_me	我发出的评论列表
	comments/to_me	我收到的评论列表
	comments/timeline	获取用户发送及收到的评论列表
	comments/mentions	获取@到我的评论
	comments/show_batch	批量获取评论内容
写入接口	comments/create	评论一条微博
	comments/destroy	删除一条评论
	comments/destroy_batch	批量删除评论
	comments/reply	回复一条评论

图 2-11　新浪微博开放 API

2. API 的命名

API 的命名务必要做到从 API 名称就能明白这个 API 的作用。在创业团队中一般就只有一两个开发人员负责后台，当开发人员要负责维护几十甚至上百个 API 时就能体会到不能“望名知 API”到底有多痛苦。

API 的命名可参考新浪微博的命名风格，例如删除新浪微博的 API“statuses/destroy”，第一个是对象，第二个是对象的操作删除。

3. API 的安全性

这点会在后面的章节“3.2 App 通信安全”中详细论述。

4. API 所返回的数据

App 客户端的主要开发语言 Java 和 Objective-C 都是强类型语言，所以怎么处理空值显得特别重要，不合理的设计很容易造成 App 的闪退。

从 App 后台的角度来说，API 返回的数据中正确值和空值的类型必须一样，举例来说，用户名的字段是{“realname”: “xxx”}，如果用户名为空，则应该返回{ “realname”: “”}。如果返回值是一个数组类型，空数据则返回一个空数组，绝对禁止返回 null 值。

对于 App 客户端，必须用全局的函数来处理所有 API 的返回数据，其目的是实现下面的机制：如果 API 的返回数据中缺少客户端需要的某个数据，App 客户端自动补上这个数据并赋予默认值。使用这个机制后，笔者团队开发的 App 大大减少了闪退的次数。

数据库设计的时候，一个合理的设计原则是所有字段都有默认值，不应该允许 Null 值，

Null 值在语言和数据库中会带来无穷的问题。当然如果产品中有 Null 值的需求，那就个别对待。产品中常见的 Null 值需求是用 Null 值表示信息未填写，例如用户的性别状态，用 “1” 表示男性，“0” 表示女性，用“Null”表示用户未填写性别。

如果 App 后台的开发语言是 PHP，会存在一个问题：PHP 中数组和字典都属于 Array，但在 Java 和 Objective-C 中这两者是不一样的，这个问题读者需要注意。

5. 图片的处理

在不同的 App 版本中，各种不同尺寸的手机中，同一张图片显示的尺寸未必一样，如果每次 App 后台都返回原图然后在 App 客户端处理，则极大浪费网络资源。如果 App 后台处理完图片才返回，则又有一个新的技术问：怎么有效保存和裁剪多种图片尺寸？

例如，App 中头像只需要返回 60×60 的尺寸，后来新版本需要返回 70×70，又出了一个新版本，需要返回 80×80，每次增加一个新的尺寸怎么在数据库上记录下来呢？笔者在刚开始设计 API 的时候没考虑这个问题，后来不得不用了一个极端的方法：每次增加新的图片尺寸就在数据库中增加一个新的字段，在这个新字段中保存新的图片路径，最后数据库的头像字段有“avatar”、“avatar_60_60”、“avatar_70_70”、“avatar_80_80”这种极度恶劣的设计。

笔者所在的团队经过不断探索，最终对图片优化的策略如下：

（1）App 客户端本地缓存图片，当缓存图片不存在才请求服务器的 API。

（2）当 App 客户端需要某种尺寸的图片，由 App 客户端通知服务端所需图片的尺寸，由服务端动态生成并缓存。

例如，App 客户端需要图片（http://www.baidu.com/img/bdlogo.gif）80×80 的尺寸，则在图片的路径加上宽和高的参数（类似于 CDN 的机制）http://www.baidu.com/img/bdlogo.gif?w=80&h=80，服务器收到这个请求解析其中的宽和高参数后，生成 80×80 的尺寸并返回给 App 客户端。

App 后台采用了这种图片处理机制后，数据库中只需一个保存原图的字段就行，其他尺寸就由客户端告诉服务端动态生成。以后无论什么尺寸的图片，数据库中都不需要记录，数据库只保存原图就行了。

注意：现在的文件云存储服务（例如七牛，又拍云等）和 CDN 都提供了这个图片的缩放功能，而且能加速文件的上传下载速度，极大地提升了 App 的用户体验，强烈推荐使用。

6. 返回的提示信息

返回信息最科学的情况是 App 后台只返回信息代码，具体的文字提示由 App 客户端决定。

如果文字信息是由 App 后台返回，则最起码要区分 2 种信息：提示用户的信息，提示 App 客户端程序员的信息。这两者的区别如下。

- 提示用户的信息是要让用户知道，提示 App 客户端程序员的信息不需要让用户知道。
- 提示用户的信息文字友好，用户不需要专业基础也能一看就知道是什么意思，提示 App 客户端程序员的信息则专业，例如告诉客户端少传了哪个参数、哪个参数有问题等。

7. 在线 API 测试文档

App 的 API 在线测试文档既是一份在线 API 文档，也是一个在线测试 API 的工具，极大地方便了双方的沟通和测试。当客户端程序员觉得某个 API 有什么问题时，双方就根据这个在线工具的结果讨论沟通。

这个 API 在线测试文档是使用 Swagger-UI 搭建的。Swagger-UI 简单而一目了然。它能够纯粹地基于 HTML+JavaScript 实现，只要稍微整合一下便能成为方便的 API 在线测试工具。项目的设计架构中一直提倡使用 TDD（测试驱动）原则来开发，Swagger-UI 在这方面更是能提供很大帮助。

Swagger-UI 的详细介绍，请查询“2.1.5 编写在线 API 测试文档”章节内容。

8. 在 App 客户端启动时调用一个 API 获取必要的初始化信息

App 客户端通过调用这个初始化 API 获取必要的信息，例如当前 App 后台最新的 App 版本，当发现本地 App 的版本已经低于最新的 App 版本时，可提示用户更新 App。当然了，很多第三方 SDK 都提供了这个提示版本更新的功能。

9. 关于 API 的版本升级问题

当 App 客户端做了大改版后可能会出现一个问题：现在的 API 已经不适应当前的 App 客户端了，这时就要考虑 API 版本的升级，同时为了兼容已经发布的 App 客户端，原来的 API 必须要保留。为了避免同一个 App 客户端中调用不同版本的 API，一般会全部升级 API 的版本，例如：原来的是“test.com/v1/statuses/destroy”，升级为“test.com/v2/statuses/destroy”。

在 API 的版本升级时，需要注意以下 2 点。

- V2 版本的 API 的 Controller 必须要继承 V1 版的 Controller，这样 V2 版本的 API 只重写需要改动的 API。
- 在线 API 测试文档中详细标明返回内容，方便 App 客户端人员的调试。

2.3 如何选择合适的数据库产品

在 App 后台开发中经常面临的问题是：用什么数据库产品存储数据？是选择 MySQL ？Redis？还是 MongoDB？

现在有这么多优秀的开源数据库产品，怎么根据业务场景来选择合适的数据库产品呢？

常用的数据库产品的优缺点又是什么？

阅读本节能帮读者解决以上的疑惑，读者再碰到选择数据库产品问题时思路会更清晰。

2.3.1 Redis，MongoDB，MySQL 读写数据的区别

数据涉及读和写这两个问题。出于性能的考虑，当然希望读和写的速度越快越好。

计算机中常见的存储设备是内存和硬盘，其特性如下。

- 内存的读取速度大概是硬盘的多倍。因此为了获得更快的读写速度，数据尽可能放在内存。
- 内存的容量有限。例如 UCloud 服务器最多只能拥有 64G 的内存，而 UCloud 服务器上的单个硬盘可高达 1000G。

Redis 的数据是存放在服务器的内存，当内存用满了后需要扩容，就只能使用 Redis 的分布式方案。为了防止断电或 Redis 程序重启造成内容数据的丢失，可调整 Redis 配置文件，按照一定的策略把数据持久化传到硬盘。

MongoDB 同时使用了硬盘和内存，其使用了操作系统提供的 MMAP（内存文件映射）机制进行数据文件的读写，MMAP 可以把文件直接映射到进程的内存空间中，这样文件就会在内存中有对应的地址，这时对文件的读写是能通过操作内存进行的，而不需要使用传统的如 fread、fwrite 文件操作方式。MMAP 的机制可参考“8.2.1 MMAP——内存文件映射”

MySQL 的数据是放在硬盘中。虽然 MySQL 也有缓存，但 MySQL 缓存的是查询的结果，而不是缓存数据。

2.3.2 Redis，MongoDB，MySQL 查找数据的区别

如果读者需要在一栋大楼里找某个房间，但是读者不知道这个房间的门牌号，只记得这个房间的门是非常特别的，那找到这个房间唯一的方式只能每层楼逐个房间找一次，比较一下房间的门和记忆中房间的门是否相同。

如果读者知道这个房间的门牌号，那很简单，直奔那个楼层就行了。

Redis 的数据是基于“键值对”存储，“键”相当于门牌号，“值”相当于房间。Redis 查找数据，每次都是直奔目标，读写速度当然快。

MongoDB 和 MySQL 中，每组数据都有一个 id（或者可以为每组数据建立索引），这个 id 或索引就相当于门牌号。

MongoDB 和 MySQL 中查找数据，有两种模式：知道 id 或索引，不知道 id 或索引。知道 id 或索引的情况就相当于知道门牌号，直奔目标就行，效率高。如果不知道 id 或索引的情况下查找数据，那就相当于在每层楼逐个房间找，效率低。

2.3.3　Redis，MongoDB，MySQL 适用场景

1. Redis 适用场景

数据读写速度快，但由于 Redis 数据只存放在服务器的内存（可采用 Redis 的分布式方案扩容），内存的价格高，所以用内存存储数据成本高。

同时由于 Redis 存放的数据必须是键值对（key-value）的形式，在读写 Redis 数据时必须要知道所读写数据的键，这点读者在使用 Redis 时需要考虑。

所以在 App 后台中，读写频率高的数据一般都会放在 Redis 中（当然这部分数据也可以同时存放于 MySQL 或者 MongoDB， Redis 中的数据是以缓存的形式存在的，当数据更新的时候，两部分都要更新以保持数据的一致性）。

例如 API 中附带了用户的身份信息，由于每次 API 操作都需要验证用户的身份信息，这些身份信息的数据存放在 Redis 中就非常合适，因为验证用户信息是个频率非常高的操作。

2. MongoDB 适用场景

MongoDB 适用于下面的场景。

- 网站数据：MongoDB 非常适合实时的插入、更新与查询，并具备网站实时数据存储所需的复制及高度伸缩性。
- 大尺寸、低价值的数据：使用传统的关系数据库存储一些数据时可能会比较贵，在此之前，很多程序员往往会选择传统的文件进行存储。
- 高伸缩性的场景：MongoDB 非常适合由数十或者数百台服务器组成的数据库。
- 存储地理坐标的数据：MongoDB 的地理坐标查询功能非常强大，例如，MongoDB 可以查找在某个矩形范围内的所有坐标，因此 MongoDB 非常适合于 LBS 应用。

但 MongoDB 不适用于下面的场景。

- 高度事物性的系统：例如银行或会计系统。传统的关系型数据库目前还是更适用于需

要大量原子性复杂事务的应用程序。例如，涉及金钱的操作，假设要转账，必须从一个账号上扣钱，再在另外一个账号上把钱转账。这个操作必须保证要么两个都完成，要么两个都不做，不能只做一个。但很遗憾，由于 MongoDB 不支持事务，所以没法保证。

- 传统的商业智能应用：针对特定问题的 BI 数据库会产生高度优化的查询方式。对于此类应用，数据仓库可能是更合适的选择。
- 需要 SQL 的问题：虽然 MongoDB 支持类似于 SQL 的查询方式，但它的查询比起 MySQL 还是有一定的差距。

（3）MySQL 适用场景

- 事物性的系统。例如，在 MongoDB 中举的转账例子。
- 需要复杂 SQL 的问题。

关于 Redis，MongoDB，MySQL 更详细的介绍，请查看本书第 6 章、第 7 章和第 8 章的内容。

2.4 如何选择消息队列软件

很多没有实际项目经验的读者对消息队列非常陌生，看着很多架构的介绍中都提到消息队列，但是不知道为什么要用消息队列？什么是消息队列？常见的消息队列产品有哪些？

本节就是为读者解决以上疑惑的。

2.4.1 为什么要用消息队列？

在某个公司里，一个管理者接到上级的任务。管理者在完成这个任务时，把这个任务分解为几个小任务，只要分别完成了这几个小任务，整个任务也就完成了。

做某个小任务时，管理者发现这个小任务需要花很多时间完成，而且这个小任务迟点完成也不影响整个任务的完成进度。于是管理者把这个小任务交给一个下属去做，自己继续完成其他的任务。

在上面的例子中，管理者就是 App 后台，下属就是消息队列，当后台系统发现完成某些小任务需要花很多时间，而且迟点完成也不影响整个任务的完成进度时，就会把这些小任务交给消息队列。

在 App 后台中，发送邮件、发送短信、推送消息等任务都非常适合在消息队列中处理。读者想想这些任务是不是都需要花比较多的时间，而且迟点完成也不影响的。把这些任务放在

消息队列中，可加快 App 后台请求的响应时间。

同时消息队列也能把大量的并发请求变成串行的请求，来减轻服务器的负担。

2.4.2　消息队列的工作流程

消息队列一般都包含 3 个角色：队列服务端，队列生产者，队列消费者。

消息队列类似于这个场景：一条产品传送带在不停地运转。在传送带的起点，工人 a 不断地把产品放在一个盒子中，然后把盒子放到传送带上，盒子被传送带传送到终点。在终点工人 b 把盒子的产品取出来进行加工处理。

在上面的场景中，不停运转的传送带就是队列服务端，在传送带起点不断放盒子的工人 a 就是队列的生产者，在传送带终点不断取盒子的工人 b 就是队列的消费者。

现在有大量开源的应用可作为消息队列的服务端，例如 RabbitMQ，ZeroMQ，Redis 等。

消息队列的工作流程如下。

App 后台把消息推入到消息队列，这里 App 后台是充当队列生产者。

守护进程（可以理解为专门处理消息的工人，即队列消费者）不断地检测消息队列中有没有新的消息，没有消息就休息一会儿再检测消息队列中有没有新的消息（这样做能避免消息队列占据过多的服务器资源），有消息的话就从消息队列取出消息，用新的线程处理相关的业务，在主进程中继续检测消息队列是否有新消息。

消息处理流程如图 2-12 所示。

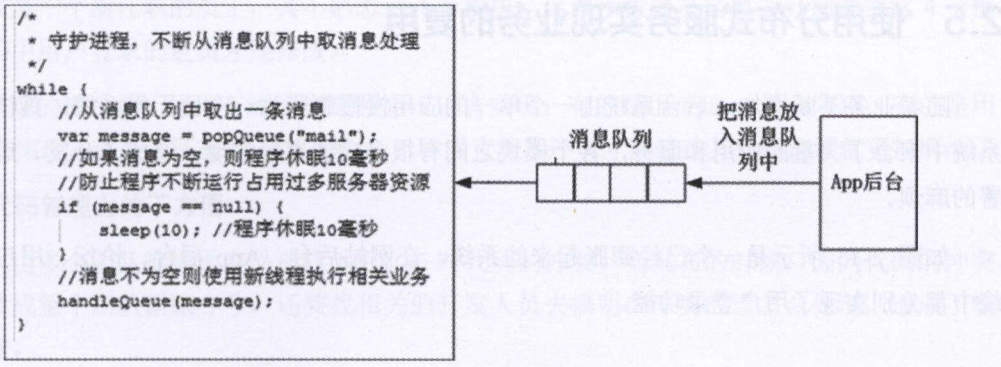


图 2-12　消息处理流程

2.4.3 常见的一些消息队列产品

1. RabbitMQ

RabbitMQ 是使用 Erlang 编写的一个开源消息队列，其支持大量的协议：AMQP、XMPP、SMTP、STOMP。也正是如此，使得它变得非常重量级，更适合于企业级的开发。RabbitMQ 同时实现了一个经纪人（Broker）架构，这意味着消息在发送给客户端时先在中心队列排队。RabbitMQ 对路由（Routing）、负载均衡（Load balance）或者数据持久化都有很好的支持。

同时 RabbitMQ 自带了一个 Web 监控界面，可方便监控队列的情况。

2. Redis

Redis 虽然是一个 key-value 系统，但其也支持队列这种数据结构，可看作是一个轻量级的消息队列。

在 App 后台架构中 Redis 被广泛使用，如果把其作为消息队列，能减少项目中的运维成本。

3. ZeroMQ

ZeroMQ 号称为最快的消息队列，尤其针对大吞吐量的需求场景。

4. ActiveMQ

ActiveMQ 是 Apache 软件基金会下的一个子项目， 类似于 ZeroMQ，它能够以代理人和点对点的技术实现队列。

2.5 使用分布式服务实现业务的复用

随着业务不断增加，后台系统由一个单一的应用慢慢膨胀为一个巨无霸系统，具体表现为系统中聚合了大量的应用和服务，各个模块之间有很多功能重复实现，造成了开发、运维、部署的麻烦。

如图 2-13 所示是一个已经膨胀起来的系统，在网站后台、App 后台、论坛、用户管理系统中都分别实现了用户登录功能。

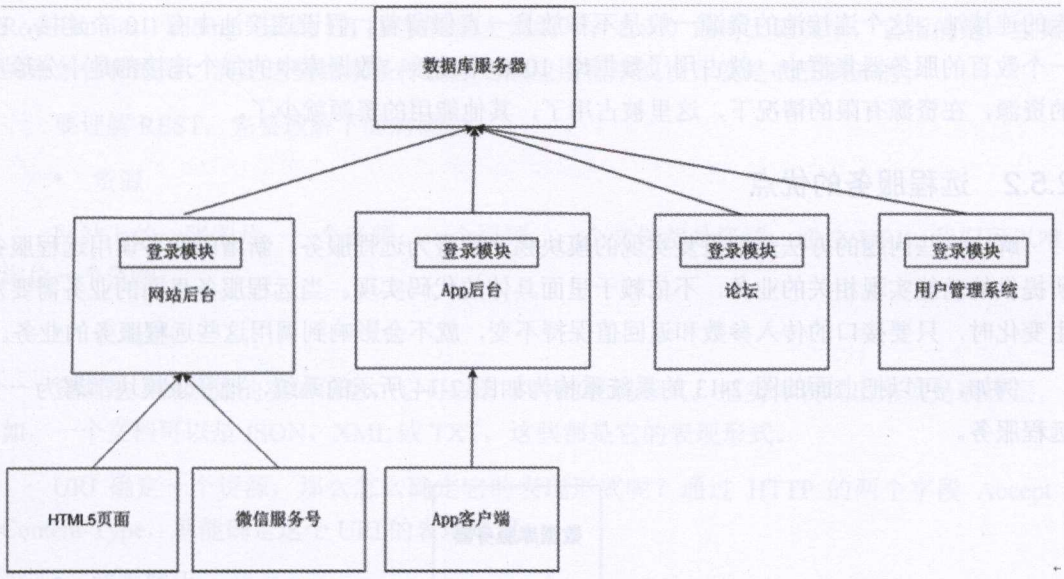


图 2-13　膨胀的系统

2.5.1　巨无霸系统的危害

1. 维护上的麻烦

系统里面有许多行代码，可能因为历史的原因同一个功能在不同的模块上重新实现了一次。

如果因为业务上的要求需要更改用户登录的规则，在图 2-13 中网站后台、App 后台、论坛、用户管理系统这 4 个模块中的用户登录代码都需要修改。

如果是一个新任职的员工，其不熟悉这 4 个模块，这表示程序员需要去分别读懂这 4 个模块的代码中用户登录的逻辑才能修改。

更悲观的情况是，如果有的模块是用 Java 语言实现，有的是用 Python 语言实现，有的是用 PHP 语言实现，这表示还需要一个了解这 3 种语言的程序员才能修改其代码，那就更加麻烦了。

2. 代码管理上的不方便

模块由多个团队维护很容易造成 MERGE 时代码的冲突。当发布的时候，因为代码的冲突，往往会造成整个团队紧张兮兮，还要找相关的开发人员去搞明白冲突应该怎么处理，造成了极大的不便。

3. 数据库连接资源的耗尽

大量应用中的重复模块会带来大量的访问，而每个应用与数据库的连接，一般是使用数据

库的连接池，这个连接池的资源一般是不释放且一直保留着。假设连接池中有 10 个连接，在一个数百的服务器集群中，就占用了数据库 1000 个连接。数据库中的每个连接都是十分珍贵的资源，在资源有限的情况下，这里被占用了，其他能用的资源就少了。

2.5.2 远程服务的优点

解决这些问题的方法是把重复实现的模块独立部署为远程服务，新增的业务调用远程服务所提供的功能实现相关的业务，不依赖于里面具体的代码实现。当远程服务里面的业务需要发生变化时，只要接口的传入参数和返回值保持不变，就不会影响到调用这些远程服务的业务。

例如，可以把上面的图 2-13 的系统重构为如图 2-14 所示的系统，把登录模块部署为一个远程服务。

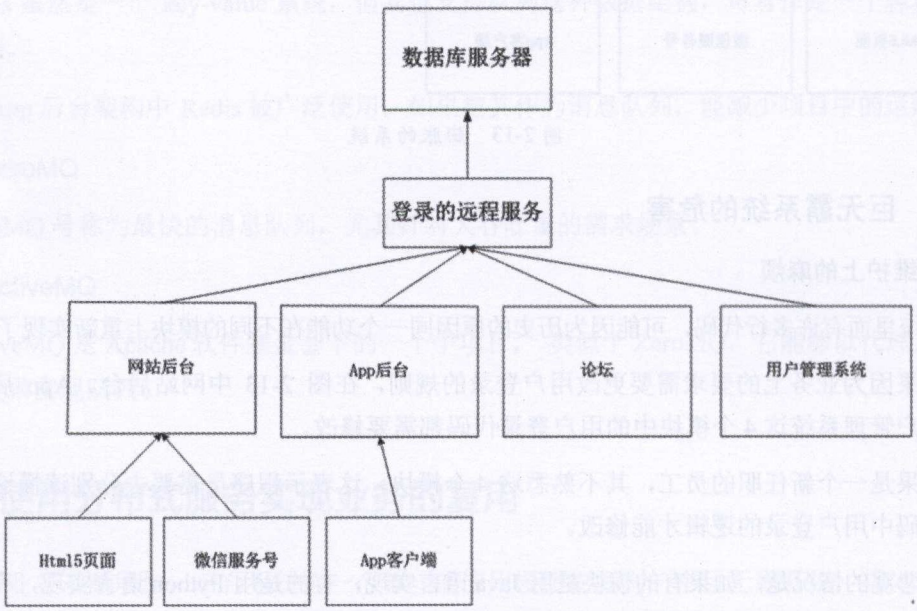


图 2-14 使用远程服务后的系统

在图 2-14 中可看到，登录模块已经独立出来，如果业务上修改登录的逻辑，只需要修改登录模块中的代码，不影响到网站后台、App 后台、论坛、用户管理系统这 4 个模块。

2.5.3 远程服务的实现

1. REST

REST（REpresentational State Transfer，表现层状态转化）这个概念首次出现是在 2000 年

Roy Thomas Fielding（其是 HTTP 规范的主要编写者之一）的博士论文中，它指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 RESTful 的。

要理解 REST，先要理解下面的概念：

- 资源

网站上的一张图片、一个音频、一个视频、一个文档都是资源。通过 URI，我们可以唯一定位一个资源。

- 表现层

资源是一种具体的实体信息，它可以有多种的表现形式。把实体表现出来就是表现层。例如，一个文档可以是 JSON、XML 或 TXT，这些都是它的表现形式。

URI 确定一个资源，那么怎么确定它的表现形式呢？通过 HTTP 的两个字段 Accept 和 Content-Type，就能确定这个 URI 的表现形式。

- 状态转化

在客户端和 App 后台交互的过程中，就已经涉及资源状态的变化。但是，HTTP 协议是无状态的协议，状态是保存在 App 后台。

客户端和 App 后台的交互通过 HTTP 协议实现。客户端需要通知服务端状态的变化，也就只能通过 HTTP 协议。具体来说是通过下面 4 个表示操作方式的动词实现。

- GET：获取资源
- POST：新增资源
- PUT：修改资源
- DELETE：删除资源

总结 REST 架构的特点。

- 每一个 URI 代表一种资源。
- 客户端和 App 后台之间，传递这种资源的某种表现层。
- 客户端通过 4 个 HTTP 动词，对 App 后台资源进行操作，实现“表现层状态转化”。

REST 设计原则中最重要的是 App 端和 App 后台之间的请求是无状态的，一个请求中必须包含理解请求所需的信息。无状态请求可使用负载均衡技术由集群中的一台服务器应答，十分适合云计算之类的环境。同时 App 端可以缓存数据以改进性能。

2. RPC

RPC（Remote Procedure Call Protocol）——远程过程调用协议，其是一种通过网络从远程

计算机程序上请求服务，而不需要了解底层网络技术的协议。它假定某些传输协议的存在，如 TCP 或 UDP，以便为通信程序之间携带信息数据。通过其可以使函数调用模式网络化。

在 OSI 网络通信模型中，RPC 跨越了传输层和应用层。RPC 使得开发包括网络分布式多程序在内的应用程序更加容易。RPC 的原理如图 2-15 所示。

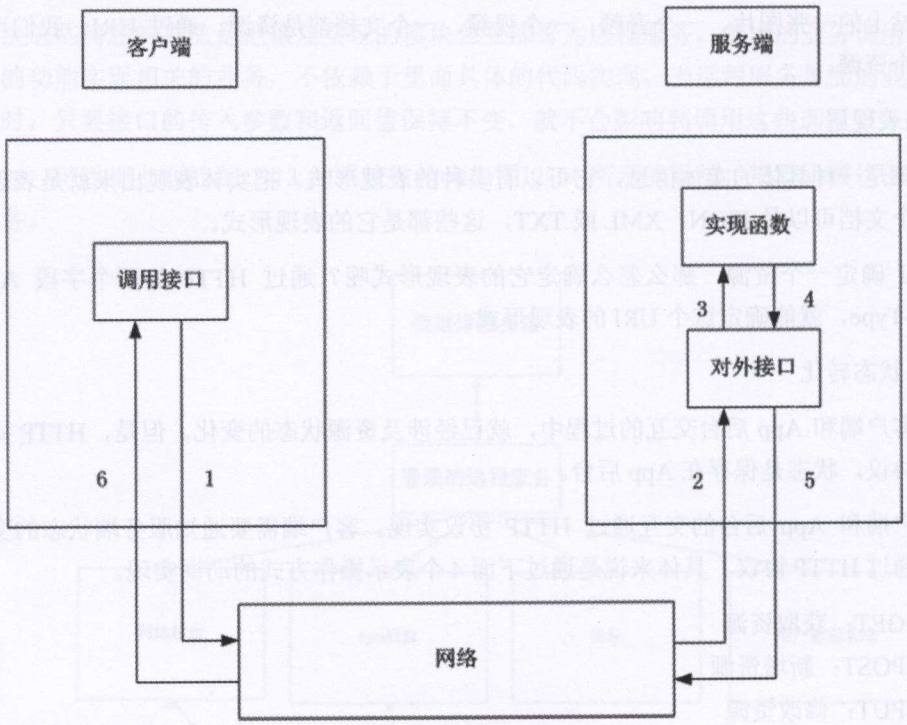


图 2-15　RPC 原理图

RPC 的调用过程如下。

（1）客户端通过接口传送参数，把参数传送到网络。

（2）服务端的对外接口通过网络接收参数。

（3）服务端把参数传递到实现函数。

（4）实现函数实现相关的功能，把返回的数据返回到对外接口。

（5）对外接口把返回数据传送到网络。

（6）客户端接口接收到网络返回的数据。

3. 开源的 RPC 库

先介绍一个轻量级的 RPC 库：Hprose（High Performance Remote Object Service Engine，http://hprose.com/）是一款先进的轻量级、跨语言、跨平台、无侵入式、高性能动态远程对象调用引擎库。其不仅简单易用，而且功能强大。

这个开源的 PRC 库已经实现了所有主流语言的服务端和客户端的 RPC，开发者只需要简单地调用就能构建 RPC 服务。

另外一个是阿里巴巴的开源的 Dubbo，其是一个分布式服务框架，致力于提供高性能和透明化的 RPC 远程调用服务和 SOA 服务治理方案。

当当网在 Dubbo 的基础上实现了如下的新功能，并将其命名为 Dubbox。

- 支持 REST 风格远程调用（HTTP + JSON/XML)。
- 支持基于 Kryo 和 FST 的 Java 高效序列化实现。
- 支持基于嵌入式 Tomcat 的 HTTP remoting 体系。
- 将 Dubbo 中 Spring 由 2.x 升级到目前最常用的 3.x 版本。
- 将 Dubbo 中的 ZooKeeper 客户端升级到最新的版本，以修正老版本中包含的 bug。

2.6　搜索技术入门

现在人们的网络生活已经离不开搜索，遇到不懂的问题，想知道的事情，搜索一下就知道答案。

App 最常见的搜索情景就是搜索用户。只有几百、几千的用户量时，可以直接用 MySQL like 这种模糊查询，但是，如果数据有几百万，甚至上千万的时候，一次 like 查询就能让数据库堵了。因此数据到了一定量级的时候，不得不考虑使用搜索技术。

2.6.1　一个简单的搜索例子

下面有三行数据。

（1）近 2 周 8 成股民亏损超 10%。

（2）满仓中国梦。

（3）股民两天亏一套三居。

例如有个需求：在上面的 3 行数据中找出包含“股民”这个关键词的数据。

按照一般的做法，就是分别查找上面的每一行数据。

第一行数据从头到尾查找一次，发现有“股民”这个关键词。

第二行数据从头到尾查找一次，没有有“股民”这个关键词。

第三行数据从头到尾查找一次，发现有“股民”这个关键词。

根据查找结果，第一、第三行数据包含“股民”这个关键词。

2.6.2 搜索技术的基本原理

按照上面的过程，每次查找都需要把每行数据从头到尾查一次。如果需要从上百万、千万的数据中查找一个关键词，读者可以想象一下效率有多低。

在搜索引擎搜索“股民”这个关键词的结果如图 2-16 所示。

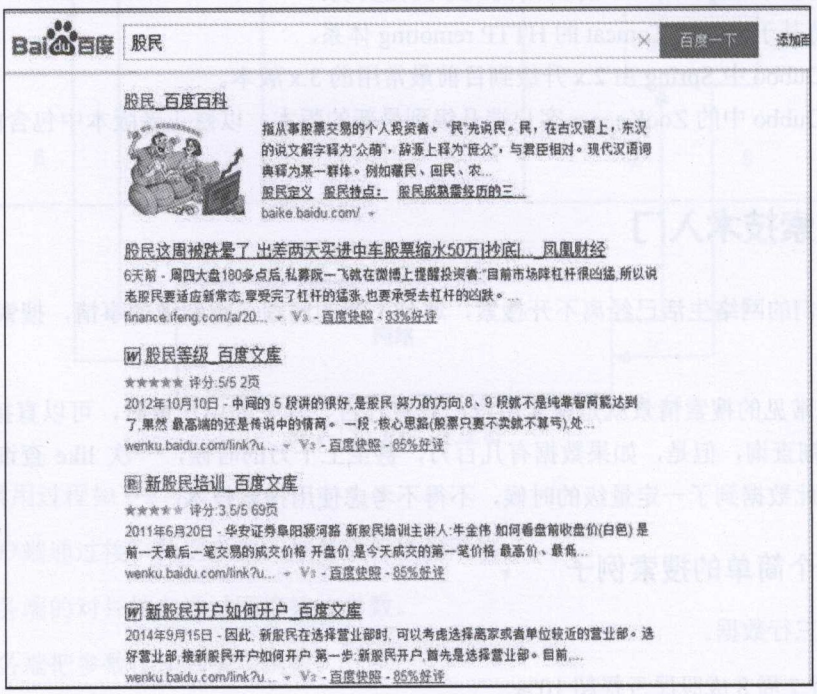


图 2-16 搜索的例子

搜索引擎的搜索结果是直接显示了所有包含“股民”这个关键字的数据，其是怎么做到在海量的信息中，快速搜索包含关键字的信息呢？

实现搜索的关键，就是分词和倒序索引。

如果知道每行数据中包含多少个关键字，然后建立一个映射表，把每个关键字出现在哪行数据中记录下来，搜索就变得很轻松。当知道一个关键字的时候，只需要查找这个映射表，找到这个关键词，根据这个关键词建立的映射关系就能查到包含这个关键词的数据。

知道每行数据中包含多少个关键字的过程，就是分词。

这里有个问题，什么是关键字？

关键字，其实就是一个词语或句子，例如，当笔者有需要的时候，“股民”可以是搜索的关键字，但是，“股”也可以是搜索的关键字，“民”也可以是搜索的关键字。什么是关键字，要看使用者的需求。因此，为了能准确分析出一行数据到底包含多少个关键字，就需要一个包含了所有词语或句子的词典，用来分析数据中有什么关键字。

建立一个映射表，把每个关键字出现在哪行数据中记录下来，这个过程就是建立倒序索引。

下面举例说明分词和建立倒序索引。

还是回到上面举例的三行数据，左边的是数据的编号，右边的是数据的内容。

（1）近 2 周 8 成股民亏损超 10%。

（2）满仓中国梦。

（3）股民两天亏一套三居。

首先，把分析上面每行数据包含多少个关键词（这里为了简化分词过程，没有把每个汉字或数字当成一个关键词，例如，“民”应该是个关键词，但为了简化分词，没有把它当成一个关键词），结果如表 2-1 所示。

表 2-1　分词的结果

数据的编号	数据中包含的关键词，每个关键词以“，”分割
1	近，2 周，8 成，股民，亏损，超，10%
2	满仓，中国，梦
3	股民，两天，亏，一套三居

下面根据表 2-1 的结果建立一个映射表 2-2，把每个关键字出现在哪行数据记录下来。

表 2-2　出现关键字的数据

关键词	关键词出现在哪行数据中
近	1
2 周	1

续表

关键词	关键词出现在哪行数据中
8 成	1
股民	1,3
亏损	1
超	1
10%	1
满仓	2
中国	2
梦	2
股民	3
两天	3
亏	3
一套三居	3

根据表 2-2，读者很容易得知 “股民”这个关键词在数据 1,3 中出现过。如果需要知道“中国”这个关键词出现在哪，通过查找表 2-2 也很容易得知其出现在数据 2 中。

在这么几行数据中，还不能体验到倒序索引的高效。但如果数据量到了上百万、千万，甚至上亿，倒序索引的效率就非常高了。

再进一步，在表 2-2 的右侧，除了记录关键词出现在哪行数据，还记录在某行数据中出现的频率，出现的位置等信息，如果读者有兴趣继续深入了解搜索引擎的技术，可阅读《这就是搜索引擎：核心技术详解》（张俊林著），本节只是简单介绍搜索引擎的基本原理。

2.6.3 常见的开源搜索软件介绍

搜索技术一点都不简单，如果要研发人员从头开始做，不知道要到哪年哪月才能用给 App 加上搜索功能。幸好，计算机高手们已经开源大量的搜索软件，只要读者会使用这些搜索软件提供的 API，就能给 App 后台加上搜索技术。下面简单介绍一下常见的搜索软件。

1. Lucene

Lucene 是一套用于全文检索和搜寻的开源程式库，由 Apache 软件基金会支持和提供。Lucene 的目的是为软件开发人员提供一个简单易用的工具包，以方便在目标系统中实现全文检索的功能，或者是以此为基础建立起完整的全文检索引擎。Lucene 提供了一个简单却强大的应用程式接口，能够做全文索引和搜寻。在 Java 开发环境里 Lucene 是一个成熟的免费开源工具。就其本身而言，Lucene 是最近几年很欢迎的免费 Java 信息检索程序库。

2. Solr

Solr 是一个高性能，采用 Java5 开发，基于 Lucene 的全文搜索软件，同时对其进行了扩展，提供了比 Lucene 更为丰富的查询语言，同时实现了可配置、可扩展并对查询性能进行了优化，并且提供了一个完善的功能管理界面，是一款非常优秀的全文搜索引擎。它对外提供类似于 Web-Service 的 API 接口，用户可以通过 HTTP 请求向搜索引擎服务器提交一定格式的 XML 文件，生成索引；也可以通过 HTTP Get 操作提出查找请求，并得到 XML 格式的返回结果。

3. ElasticSearch

ElasticSearch 是一个基于 Lucene 的搜索服务器。它提供了一个分布式多用户的全文搜索引擎，基于 RESTful Web 接口。ElasticSearch 是用 Java 开发的，并作为 Apache 许可条款下的开放源码发布，是第二流行的企业搜索引擎。

4. Sphinx

Sphinx 是一个基于 SQL 的全文检索引擎，其结合 MySQL、PostgreSQL 做全文搜索，可以提供比数据库本身更专业的搜索功能，使得应用程序更容易实现专业化的全文检索。Sphinx 特别为一些脚本语言设计搜索 API 接口，如 PHP、Python、Perl、Ruby 等，同时为 MySQL 也设计了一个存储引擎插件。

5. CoreSeek

CoreSeek 是一款中文全文检索/搜索软件，以 GPLv2 许可协议开源发布，基于 Sphinx 研发并独立发布，专攻中文搜索和信息处理领域，适用于行业/垂直搜索、论坛/站内搜索、数据库搜索、文档/文献检索、信息检索、数据挖掘等应用场景，用户可以免费下载使用。

CoreSeek 曾经在笔者架构的两个 App 后台中深度使用过，配置简单，性能高效，整合了 Sphinx 和中文分词，可以快速完成搜索模块的开发。但最大的缺点是稳定版不支持实时索引，测试版虽然是支持实时索引，但一直都没发布稳定版本。

Coreseek 有两个核心模块 Indexer 和 Search。

- Indexer：负责从 MySQL 拉取数据源，把数据源分词，建立索引。
- Search：搜索模块。

CoreSeek 的原理如图 2-17 所示。

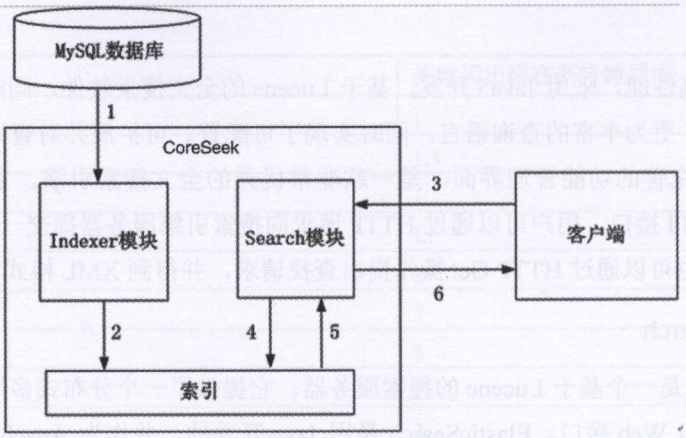


图 2-17　CoreSeek 原理图

在图 2-17 中，CoreSeek 工作流程如下：

（1）Indexer 模块从 MySQL 中拉取数据。

（2）Indexer 模块用经过中文分词后的数据建立索引。

（3）客户端向 Search 模块发起搜索请求。

（4）Seach 模块查找索引中的数据。

（5）Seach 模块得到索引中符合要求的数据的 id 等数据。

（6）把数据返回给客户端。

另外笔者分享一个经验：搜索的时候，有的用户是通过直接输入拼音来代替汉字搜索的，如图 2-18 所示。

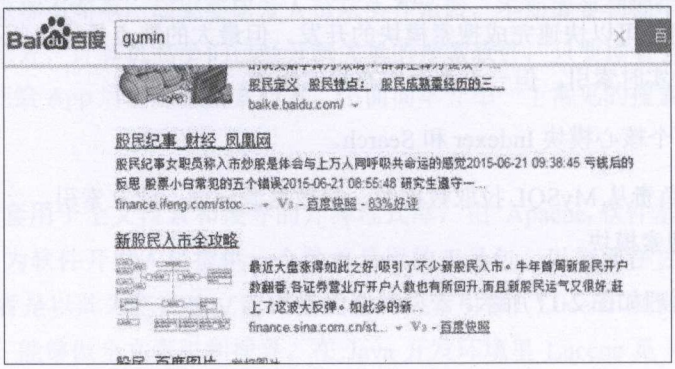


图 2-18　输入拼音代替汉字

这种情况就要在保存关键字的同时也保存关键字的拼音，建立索引时也使用拼音建立索引，就能实现用拼音搜索。

参考资料如下。

（1）http://baike.baidu.com/link?URL=rNBW3tzH-oJYeBoPSUvWZPGz-stIkE5zFQsjAtV234-HFFPJKyeyr3dJjJrbZKRSCBg2NGZv-lA7DFqHF5XBEoq

（2）http://baike.baidu.com/link?URL=C92bKEtkJtap8FfRjpSX4m5-yGE1Dn6O-00FRV5RwLe-EOkJ6FIvfl7amUuYceb-5jOD3Zn0Oy1_1vh7LG0RXK

（3）http://baike.baidu.com/link?URL=xH1aipHlRiiq3JduGb8J8aT7qpYxs1rVDuvUQe76z0WLDZ-vuPFuI8Y7pbthYyiUZyyAB5wUxFzJqs5oAnRh5PHPO7XYvdFSvuV5JlNVuD33

（4）http://www.coreseek.cn/

2.7　定时任务

App 后台开发经常需要执行一些定时任务，例如：定期清理一下项目产生的垃圾文件，或者要某段时间执行一些业务逻辑等，都需要使用到定时任务。下面介绍一下 Linux 的定时任务和用开发语言实现的定时任务两种方案。

2.7.1　Linux 定时任务 Crontab

在 Linux 使用“Crontab–e”就能创建定时任务。定时任务写入到/var/spool/cron/中，注意是写入到用户的那个文件中，例如，用户 Jeff 的定时任务会写入到/var/spool/cron/jeff。千万不能使用 vi 直接编辑这个文件，因为直接编辑不能检查语法错误，使用 Crontab 编辑是能检查语法错误。

Crontab 命令的语法如下。

```
crontab [-u username] [-l|-e|-r]
参数:
-u : 只有 root 才能进行这个任务，编辑某个用户的 crontab
-e : 编辑 crontab 的工作内容
-l : 查阅 crontab 的工作内容
-r : 移除所有的 crontab 的工作内容。
```

Crontab 的命令格式如表 2-3 所示。

表 2-3 Crontab 的命令格式

代表意义	分钟	小时	日期	月	周	命令
范围	0-59	0-23	1-31	1-12	0-7	命令

数字栏特殊字符的含义如表 2-4 所示。

表 2-4 数字栏特殊字符的含义

特殊的符号	含义
*	任何时刻都接受，例如：* * * * * cmd 表示每分钟都运行 cmd
,	表示有多个时间段，例如：2,4 * * * * cmd 表示第 2，第 4 分钟都运行 cmd
-	表示时间间隔，例如：2-4 * * * * cmd 表示第 2 至第 4 分钟每分钟都运行 cmd
/n	表示隔 n 个时间单位，例如*/5 * * * * cmd 表示每隔 5 分钟运行 cmd

按照官方的文档，运行定时任务的最少单位是分钟，如果需要运行秒级的定时任务，应该怎么办呢？

一个取巧的方法如下。

```
* * * * * cmd
* * * * * sleep 20; cmd
* * * * * sleep 40; cmd
```

上面就是每 20 秒运行一次 cmd 的 Crontab 的做法。

2.7.2 在后台轻松管理各种各样的定时任务

在项目初期，需要运行的定时任务比较少，用 Linux Crontab 管理定时任务还没太大问题，随着项目的不断推进，慢慢发现了 Crontab 的不足：

- 当需要执行的定时任务有上百个的时候，Crontab 的管理形式太落后了。
- 需要执行秒级的定时任务时很不方便。
- 没有一个统一的后台查看各个定时任务的状态，例如，哪些定时任务执行成功了，哪些定时任务执行过程中有异常，异常信息是什么等。

因此针对上面的问题，后台需要引入新的定时任务框架：Java 下的 Quartz 或者 Python 下的 APScheduler。

Quartz 是 OpenSymphony 开源组织的一个开源作业调度框架，它可以与 J2EE、J2SE 应用程序相结合，也可以单独使用。

APScheduler 是基于 Quartz 的一个 Python 定时任务框架，实现了 Quartz 的所有功能，使用起来十分方便。

APScheduler 实现了以下的功能。

- 通过 RAM、MySQL、MongoDB 文件，持久化存储定时任务。
- 支持秒级的定时任务。
- 支持基于日期、固定时间间隔和 Crontab 类型的定时任务。

1. APScheduler 的安装

使用 easy_install 安装：

```
easy_install apscheduler
```

或者下载源码后安装：

```
Python setup.py install
```

2. 一个创建定时任务的例子

下面的例子演示了每 3 秒运行一次定时任务

```
from datetime import datetime
import time
from apscheduler.scheduler import Scheduler

def tick():
    print('Tick! The time is: %s' % datetime.now())

if __name__ == '__main__':
    scheduler = Scheduler()
    scheduler.add_interval_job(tick, seconds=3)
    print('Press Ctrl+C to exit')
    scheduler.start()
    # This is here to simulate Application activity (which keeps the main
    # thread alive).
    while True:
        print('This is the main thread.')
        time.sleep(2)
```

更详细的 APScheduler 的用法，请参考 APScheduler 的文档。

第 3 章
App 后台核心技术

App 是近几年才发展起来的，由于 App 客户端的特性，因此 App 后台的技术实现和一般的 Web 后台是有区别的。

3.1 用户验证方案

App 操作中经常涉及用户登录操作，用户登录就需要使用用户名和密码。为了安全起见，在登录的过程中暴露密码的机会越少越好。

登录过程中怎样才能最大程度地避免泄露用户的密码的可能呢？

用户登录后，App 后台怎么去验证和维持用户的登录状态呢？

本节提供了一套用户登录的解决方案以供读者参考。

3.1.1 使用 HTTPS 协议

避免信息的泄露，最基本的方案是所有涉及安全性的 API 请求都必须使用 HTTPS 协议。

HTTPS 协议是“HTTP 协议”和“SSL/TLS 协议”的组合。

SSL 是“Secure Sockets Layer”的缩写，中文称为“安全套接层”，其是 20 世纪 90 年代中期由网景公司设计的。原来在互联网上使用的 HTTP 协议是明文，存在很多缺点（比如传输内容会被偷窥和篡改），发明 SSL 协议是为了解决这些问题。到了 1999 年，SSL 协议因为其

应用广泛已经成为互联网上的事实标准，IETF 就在 1999 年把 SSL 协议标准化。SSL 协议标准化之后的名称改为 TLS（Transport Layer Security）协议，中文称为“传输层安全协议”。习惯上把这两者并列称呼（SSL/TLS），因为这两者可以视作同一个东西的不同阶段。

可以把 HTTPS 大致理解为“HTTP over SSL”或“HTTP over TLS”。其是一个安全通信通道，基于 HTTP 开发，用于在客户计算机和 App 后台之间交换信息。其使用安全套接字层(SSL)进行信息交换，简单来说其是 HTTP 的安全版。HTTPS 实际上应用了安全套接字层（SSL）作为 HTTP 应用层的子层。

HTTPS 的模型如图 3-1 所示。

读者看看支付宝涉及登录和支付的页面，URL 都是以 HTTPS 开头，这就意味通信是使用 HTTPS。国内主流开放平台的 API，例如新浪微博、腾讯等，API 请求都是以 HTTPS 开头。

HTTPS 是业界常用的安全协议，支付宝登录的页面就是使用了 HTTPS 协议，如图 3-2 所示。

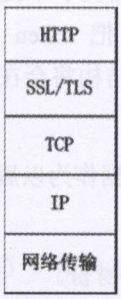


图 3-1　HTTPS 的模型

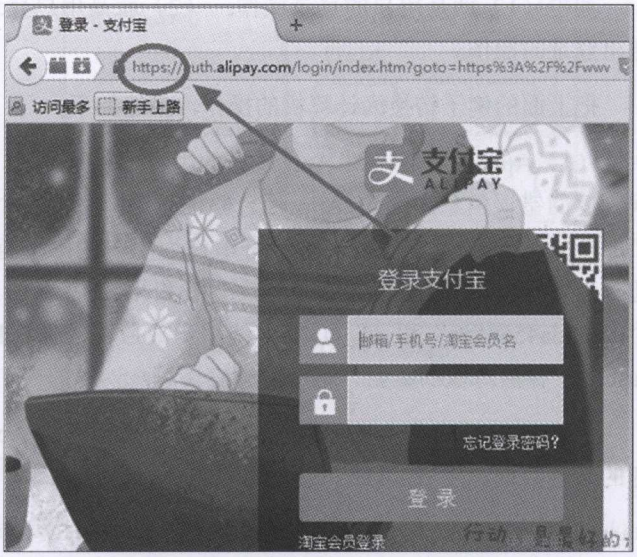


图 3-2　支付宝使用 HTTPS 的登录界面

3.1.2　基本的用户登录方案

传统 Web 网站使用 Cookie+Session 保持用户的登录状态，那么在 App 后台怎么实现类似的功能呢？在 App 后台怎么避免每次验证用户身份都需要传输用户名和密码呢？

解决上面的问题，可以参考下面例子。

把 App 后台想象为一个房间，里面有个房间管理员。同时房间门有一把锁，这把锁有两种打开方式。

- 输入了这把锁上注册的用户名和密码。
- 用房间管理员提供的钥匙。

用户进入这个房间的流程如下。

（1）用户第一次输入锁上注册的用户名和密码打开这把锁后进入房间，找到房间管理员，让其提供一把钥匙。

（2）以后用户每次需要进入这个房间用这把钥匙就行，不用担心旁边有人偷看到自己的用户名和密码，从而导致用户名和密码的泄露。

（3）用户决定一段时间内不再进入这个房间，又怕钥匙被偷，进入房间后把钥匙还给管理员，让管理员把钥匙销毁。

其中（1）就是用户的登录操作；（2）就是用户登录后验证身份的操作；（3）就是用户退出登录的操作。

把上面的例子转换为计算机的操作，描述如下。

（1）App 后台接收到 App 发送的用户名和密码后，验证用户名和密码是否正确。如果错误则返回错误信息。如果 App 后台验证正确，生成一个随机的不重复的 token 字符串（例如“daf32da456hfdh”），token 字符串作为用户的唯一标识（token 就是上面例子中提到的钥匙）。在 Redis 中建立 token 字符串和用户信息的对应关系，例如，把 token 字符串“daf32da456hfdh”和 id 为“5”的用户对应。注意：Redis 的 hash 数据结构将会在“7.2.2 hash——存储对像的数据”这节中详细讲解。

（2）App 后台把 token 字符串和用户信息返回给 App，App 保存这些数据作为以后身份验证的必备数据。生成 token 的流程如图 3-3 所示。

（3）需要验证用户身份的操作必须要把 token 字符串传给 App 后台验证身份。

例如，“test.com/user/update”是更新用户的信息的 API，这个 API 必须验证用户身份，当调用 API“test.com/user/update”时，把 token 字符串“daf32da456hfdh”附在 URL 上，变成“test.com/user/update?token=daf32da456hfdh”。

当 App 后台接收到这个 API 请求时，权限设置中要求验证用户身份，于是取出参数中 token 的值“daf32da456hfdh”，在（1）中建立的 token 字符串和用户信息的对应关系中查找，

如果没发现这个 token 值，则返回验证失败的信息，如果发现这个 token 值，则获取这个用户的信息，进行相关的更新操作。

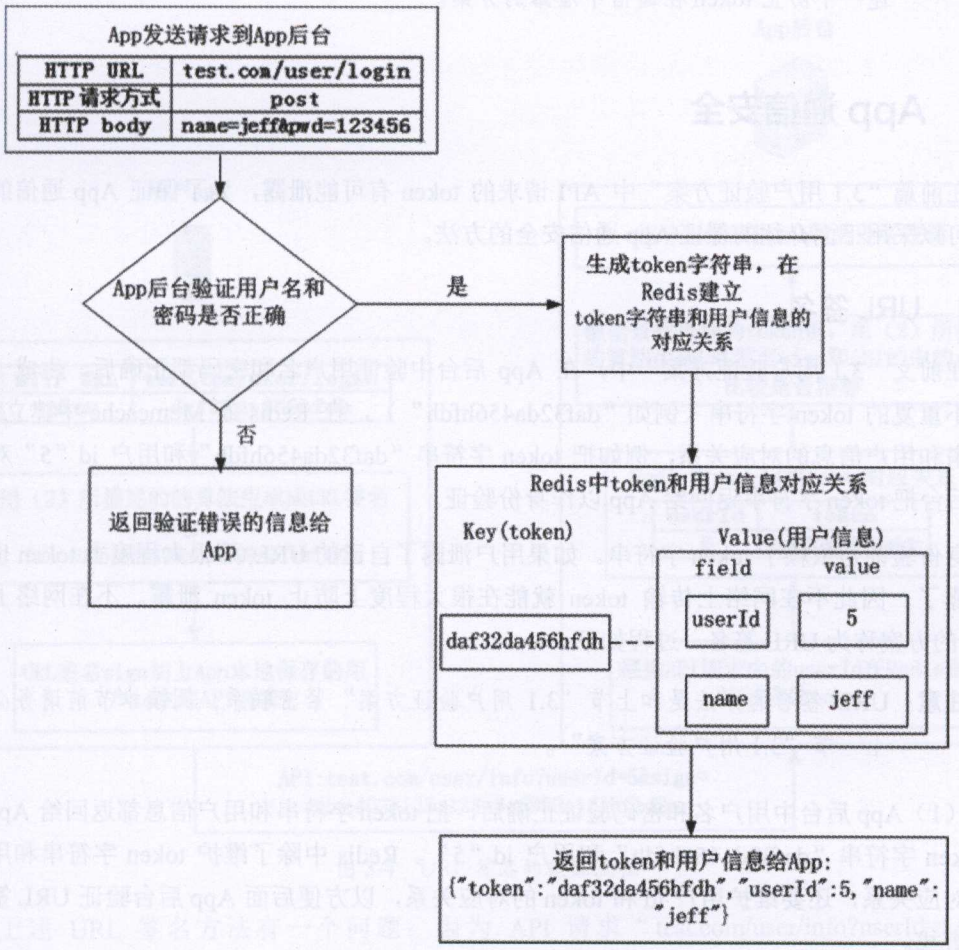


图 3-3　生成 token 的流程

（4）当用户退出登录时，通过调用退出登录的 API，让 App 后台把这个用户对应的 token 字符串删除。

退出登录的 API“test.com/user/logout”也要验证用户身份，调用时需要加上 token 变为“test.com/user/logout?token=daf32da456hfdh”。当 App 后台接收到退出登录的 API 请求时，在（1）中建立的 token 字符串和用户信息的对应关系中查找 token 字符串是否存在，token 字符串存在就把 token 和用户信息删除。

注意：这个方案并不是十分安全，身份验证依赖 token 字符串。如果用户泄露了 URL，那 token 也泄露了，这相当于钥匙被黑客复制了一份。在“3.2 App 通信安全”中将描述一个防止 token 在通信中泄露的方案。

3.2 App 通信安全

在前篇“3.1 用户验证方案”中 API 请求的 token 有可能泄露，为了保证 App 通信的安全，读者可以采用下面介绍的保证 App 通信安全的方法。

3.2.1 URL 签名

在前文“3.1 用户验证方案”中，在 App 后台中验证用户名和密码都正确后，生成一个随机的不重复的 token 字符串（例如“daf32da456hfdh”），在 Redis 或 Memcache 中建立 token 字符串和用户信息的对应关系，例如把 token 字符串“daf32da456hfdh”和用户 id“5”对应，App 后台把 token 字符串返回给 App 以作身份验证。

身份验证是依赖于 token 字符串。如果用户泄露了自己的 URL，那很大程度上 token 也被别人泄露了。因此不在网络上传输 token 就能在很大程度上防止 token 泄露。不在网络上传输 token 的方案称为 URL 签名，过程如下：

注意：URL 签名的方法是和上节“3.1 用户验证方案”紧密联系，阅读本节前请务必阅读上一节“3.1 用户验证方案”。

（1）App 后台中用户名和密码验证正确后，把 token 字符串和用户信息都返回给 App，例如 token 字符串“daf32da456hfdh”和用户 id“5”。Redis 中除了维护 token 字符串和用户信息的对应关系，还要维护用户 id 和 token 的对应关系，以方便后面 App 后台验证 URL 签名时快速查找。

（2）假设 API 请求为“test.com/user/info”，App 用 token 字符串“daf32da456hfdh”和 URL 的 md5 值作为 URL 签名 sign。

```
sign=md5("test.com/user/info?token=daf32da456hfdh")= 6dac4026135a123a3cf809a1f1bf1d2a
```

于是，API 请求加上 URL 签名 sign 和用户 id 后如下所示。

```
test.com/user/info?userId=5&sign= 6dac4026135a123a3cf809a1f1bf1d2a
```

这样 token 就不需要附在 URL。

（3）App 后台接收到这个 URL 后，用（2）所描述的算法生成签名和 sign 参数对比，如

果发现相等就表示验证通过，继续执行这个 API 的其他逻辑。

URL 签名的验证流程如图 3-4 所示。

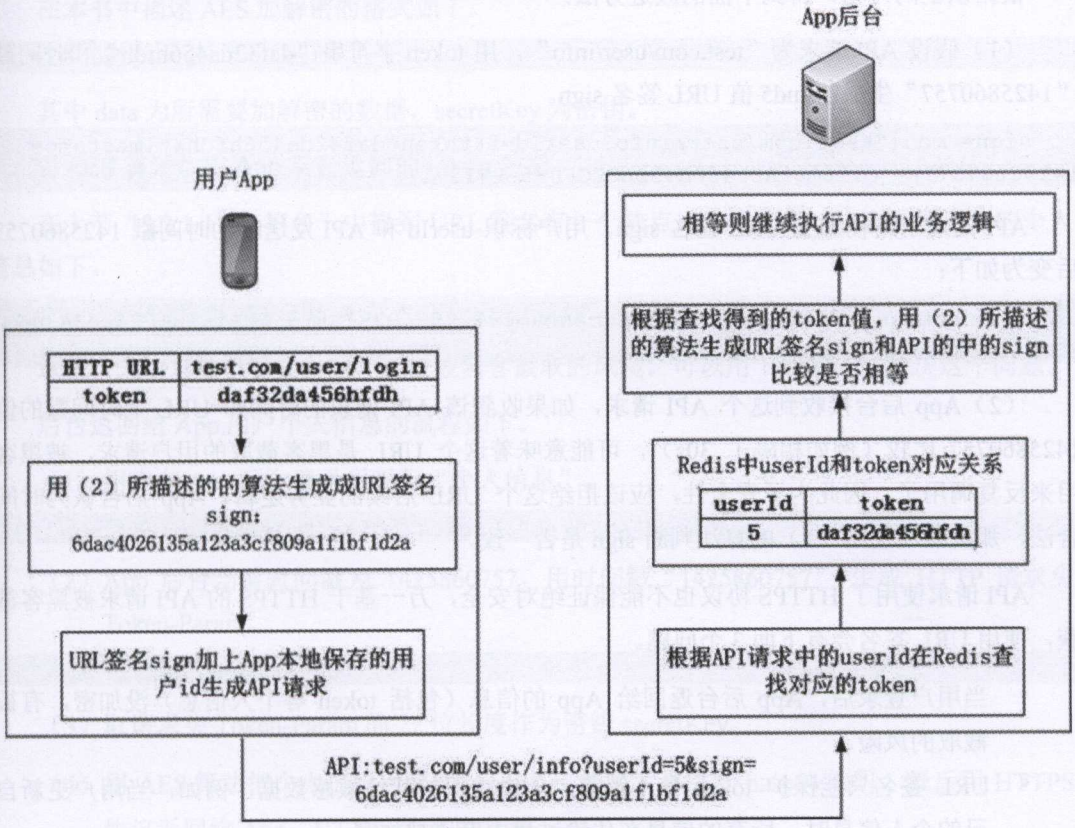


图 3-4　URL 签名的验证流程

上述 URL 签名方法有一个问题：因为 API 请求“test.com/user/info?userId=5&sign=6dac4026135a123a3cf809a1f1bf1d2a”没有过期时间，假设黑客截获了这个 API 请求的 URL，就能反复调用这个 URL。

改进方法是在传递的参数中增加时间戳，当 App 后台发现这个时间戳相隔当前时间很久的，就判断这个 URL 已经失效。

App 端使用时间戳的一个问题是：App 端的时间有可能和 App 后台的时间不一致。保证 App 端时间和 App 后台时间同步的方法为：App 每次启动时通过 API 获取 App 后台的时间，保存 App 端时间和 App 后台的时间差，以后 App 端用这个时间差调整其生成的时间戳。例如，某一刻 API 获取 App 后台的时间是 1444692778， App 端时间是 1444692775，两者的时间差是

3。当 App 在本地时间 1425860754 准备向 App 后台发送 API 请求时，加上时间差 3，即可得到 App 后台的时间为 1425860754+3=1425860757，1425860757 即为参数中传递的时间戳。

根据以上的讨论，得到下面的改进方法：

（1）假设 API 请求为“test.com/user/info”，用 token 字符串“daf32da456hfdh”和时间戳“1425860757”生成的 md5 值 URL 签名 sign。

```
sign= md5("test.com/user/info?userId=5&token=daf32da456hfdh&timestamp=
1425860757")= C116161A6F430343B6CECF08562F1371
```

API 的请求路径加上 URL 签名 sign，用户标识 userId 和 API 发送时的时间戳 1425860757 后变为如下：

```
test.com/user/info?userId=5&timestamp=1425860757&sign=c11616A6F430343B6C
ECF08562F1371
```

（2）App 后台接收到这个 API 请求，如果收到该 API 请求的时间和 URL 上时间戳的值 1425860757 比较（例如相隔了 30s），可能意味着这个 URL 是黑客截取的用户请求，被黑客用来反复调用了，因此为了安全性，应该拒绝这个 URL 后续的业务逻辑。App 后台认为时间合法，那就继续使用（1）的算法判断 sign 是否一致。

API 请求使用了 HTTPS 协议也不能保证绝对安全，万一基于 HTTPS 的 API 请求被黑客截获，使用 URL 签名会有下面 3 个问题。

- 当用户登录后，App 后台返回给 App 的信息（包括 token 等个人信息）没加密，有被截取的风险。
- URL 签名只能保护 token 值不泄露，但没法保护其他敏感数据，例如，当用户更新自己的个人信息时，所有的信息在传输过程中应该被加密。
- URL 被黑客截获后还是能在一段时间内调用（假设 App 后台认为有效的间隔时间是 30s）。

使用“3.2.2 AES 对称加密”可以解决上面提出的 3 个问题。

3.2.2 AES 对称加密

1. 对称加密的原理

采用单钥密码系统的加密方法，同一个密钥可以同时用作信息的加密和解密，这种加密方法称为对称加密，也称为单密钥加密。

假设有原始数据“1000”，把 1000 加 5 就得到密文“1005”，得到密文“1005”后减 5 就得到原始数据“1000”。把原始数据加 5 就是加密算法，把密文减 5 就是解密算法，密钥就

是 5。

本节所用的是 AES 这种通用的对称加密算法。

在本书中描述 AES 加解密的格式如下。

```
AES 加密(data, secretKey)
```

其中 data 为所需要加解密的数据，secretKey 为密钥。

2. AES 算法加密 App 后台返回的 token 数据

在上节“3.2.1 URL 签名”中提到 URL 签名有一个缺点：当用户第一次登录后用户的个人信息如下。

```
{"userId":5,"name":"jeff","token":"daf32da456hfdh }
```

这些个人信息是以明文返回，存在被黑客截取的风险。可以用下面的方法解决这个问题。

后台返回给 App 用户个人信息的流程如下。

（1）用户在 App 后台登录后得到“个人信息”。

```
{"userId":5,"name":"jeff","token":"daf32da456hfdh }
```

（2）App 后台当前时间戳为 1425860757，用时间戳“1425860757” 生成 HTTP 请求头 Token-Param。

```
Token-Param: 1425860757
```

（3）取请求头 Token-Param 的 22 位长度作为密钥 secretKey。

（4）用 AES 算法把个人信息用密钥 secretKey 加密，再进行 base64 编码，最后用 HTTPS 协议返回给 App。HTTPS 协议内容如下。

HTTP URL	https://test.com/API/login
HTTP 返回方式	post
HTTP 返回头	Token-Param: 1425860757
HTTP body	Base64Encode（AES 加密（个人信息，请求头 Token-Param 的 22 位长度））

客户端解密 App 后台返回的内容流程如下。

（1）取 HTTPS 协议中 HTTP 请求头 Token-Param 的值的 22 位长度作为密钥 secretKey。

（2）把 HTTP body 的数据先用 base64 算法解码，用 AES 算法把解码后的数据用密钥 secretKey 解密，得到用户的个人信息。

```
{"userId":5,"name":"jeff","token":"daf32da456hfdh }
```

3. AES 算法加密请求过程中所有的敏感数据

URL 签名有一个缺点：URL 签名只能保护 token 值却没法保护其他敏感数据，例如，当更新用户昵称时，用户数据在传输过程中应该是被加密的。AES 同时加密了两方面的数据。

- token
- 用户数据

下面以加密用户昵称为例子说明。

客户端加密更新用户昵称 API 请求的流程如下。

（1）客户端昵称数据。

```
{"nickName":"jeff"}
```

（2）客户程序中可得到当前时间戳：1425860767。

（3）用时间戳“1425860767”生成 HTTP 请求头 Token-Param。

```
Token-Param：1425860767
```

（4）取请求头 Token-Param 的 22 位长度作为密钥 secretKey，用 AES 算法把 token 用密钥 secretKey 加密，再用 base64 编码得到新的 HTTP 请求头 Token-Data。

```
Token-Data：Base64Encode(AES 加密(token, secretKey))
```

（5）用 token 作为密钥 secretKey2，用 AES 算法把昵称数据用密钥 secretKey2 加密，再用 base64 编码得到 HTTP body。

```
Base64Encode(AES 加密(昵称数据, token))
```

（6）最后 App 发送给 App 后台的 HTTPS 请求如下。

HTTP URL	https://test.com/API/user/update
HTTP 请求方式	post
HTTP 请求头	Token-Param：1425860767
	Token-Data：Base64Encode(AES 加密(token, 请求头 Token-Param 的 22 位长度))
HTTP body	Base64Encode(AES 加密(昵称数据, token))

后台解密更新用户昵称数据的流程。

（1）取 HTTPS 协议中 HTTP 请求头 Token-Param 的 22 位长度作为密钥 secretKey。

（2）把 HTTP 请求头 Token-Data 的数据先用 base64 算法解码，用 AES 算法把解码后的数据用密钥 secretKey 解密，得到 token。

（3）把 HTTP body 的数据先用 base64 算法解码得到 data，再以 token 为密钥 secretKey2，用 AES 算法把 data 用密钥 secretKey2 解密得到昵称为“jeff”。

注意：以上 Token-Param 和 Token-Data 都是笔者自定义的 HTTP 请求头，是因为笔者个人习惯才这样命名。

AES 算法中的密钥不是固定不变的，文中举例取请求头 Token-Param 的 22 位长度作为密钥，各位读者可以根据项目的实际情况自行设计密钥。

3.2.3　更进一步的通信安全

在上节“3.2.2　AES 对称加密”中提及的加密方法也不是万能的，如果 App 被黑客反编译了，黑客就知道整个通信的加密过程（知道使用的加密算法和如何获取密钥），如果又同时截取了 API 的请求数据，那么整个安全措施就无效。

下面是更进一步的加密措施。

- 使用自定义的通信协议传输敏感信息。
- 使用 RSA（非对称加密算法）加强通信的安全性。
- 使用梆梆加固、爱加密等第三方工具对 App 进行加密。
- 涉及到特别敏感的信息（例如支付密码），每次都需要用户输入支付密码确认，支付密码永远不在 App 端保存。
- 使用自主开发的输入控件输入敏感信息，支付宝支付的输入框就是一个典型的例子，如图 3-5 所示。

图 3-5　支付宝支付的输入框

3.3 短信服务

市场上大多数的 App 产品为了获取用户的社交关系，需要用户使用手机号注册。用手机号注册涉及一个发送短信验证码的问题。App 后台怎样发送短信呢？怎样才能确保短信服务的稳定性呢？本节为读者解开以上的疑惑。

3.3.1 App 后台发送短信简介

App 后台发送短信是个很特殊的功能，其特殊性主要体现在以下 3 点。

- 发送短信的功能没有任何开源软件可以实现。
- 发送短信只能依靠短信平台。
- 短信的到达率和延时是 App 后台研发人员无法控制的。

发送短信到用户手机必须通过运营商（移动/联通/电信等），运营商提供了短信通道作为短信发送的接口，通过短信通道可以给指定号码发送短信。

但是购买短信通道的费用非常高（百万元以上），购买资格的审核严格，使用的限制也很多，而且短信通道有被运营商封的风险。

基于短信通道的以上特点，市场上出现了短信平台这个角色。

短信平台购买了运营商的短信通道后，把一个通道给一批企业共同使用，让企业按发送短信量付费，从而降低企业发送短信的成本；同时，短信平台也初步审核短信的内容，避免触发短信通道的使用限制。

App 后台购买了短信平台的短信服务后，就能通过短信平台的 API 接口发送短信。

App 后台发送短信的流程如图 3-6 所示。

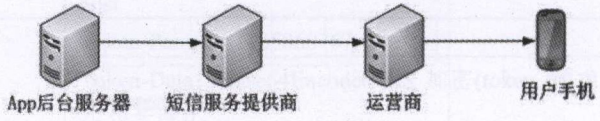


图 3-6 发送短信的流程

3.3.2 选择短信平台

从“3.3.1 App 后台发送短信简介”中可知，发送短信的关键是选择短信平台。选择短信平台主要考虑以下 2 个方面。

- 价格因素，发送一条短信的价格一般在 4 分钱到 8 分钱之间。

- 短信的到达率和延时。

笔者的手机以前经常收到各种乱七八糟的短信，例如，发票、个人贷款等。在 2013 年底，短信整顿服务，那次整顿中一大批小的短信平台倒闭了，笔者那时使用的短信平台也受到了很大的影响，发送短信到移动、联通的短信没什么问题，但发送到电信的短信，居然延迟了 2 个小时！

为了解决这个问题，笔者试了不下 5 家短信平台，后来觉得 UCloud 的监控短信到达率非常高，于是把 UCloud 的监控短信服务当成了验证码短信服务，笔者还因此把 UCloud 发送短信的 Python SDK 改写成 PHP SDK 使用。用了 UCloud 的短信一段时间后到达率变低了，又找了一家发送费用很贵（好像是 8 分钱一条短信）的短信平台才解决这个问题。

记得那段时期笔者的一个开发者朋友抱怨实在找不到可靠的短信平台，其公司就决定把 App 中的短信验证码功能去掉，用户随便输入手机号都能注册。

各位开发者在选择短信平台时一定要先试用，短信平台可以提供给开发者一定免费数量的短信，在下面的网址中 apistore.baidu.com/astore/classificationservicelist/39.html 列举了一些短信平台，读者可以试用上面的服务。记住，一定要亲自试用短信平台，别人推荐的平台，可能只是暂时有效，毕竟事物是不断变化的。

3.3.3 建立可靠的短信服务

短信服务是 App 后台中最不可控的服务，万一所使用的短信平台因为各种因素变得不稳定，那么用户就可能收不到短信或隔了很久才收到 App 后台所发送的短信。现在大多数 App 的用户注册要求验证手机号，用户收不到短信就意味着没法注册，这对 App 运营的影响极大。

为了把短信服务的风险降到最低，笔者推荐最可靠的做法是 App 后台必须要接入最少两个短信平台，当前使用的短信平台变得不可靠时，立刻切换到另外一个短信平台发送短信，主要做法是通过配置文件控制 App 后台使用哪个短信平台。

首先在配置文件中配置当前使用的短信平台，例子如下。

```
配置文件 App.conf 中关于短信平台的配置：
[sms]
currentSmsPlat=短信平台 a
```

在发送短信的逻辑中，根据配置文件确定使用的短信平台，伪代码如下：

```
/*
* 发送短信的逻辑
*/
//读取配置文件中设置使用的短信平台
```

```
smsPlat = readConf("sms: :currentSmsPlat");
if (smsPlat == "短信平台 a") {  //判断使用哪个短信发送短信
    /* 使用短信平台 a 发送短信 */
} elseif (smsPlat == "短信平台 b") {
    /* 使用短信平台 b 发送短信 */
}
```

3.4 处理表情的一些技巧

App 中文字夹带表情是个很常见的现象，甚至一些 40 多岁的大叔级用户也喜欢在自己的昵称中夹带表情，在产品运营中发现这个现象，彻底颠覆了笔者的世界观。

App 后台处理表情这个业务笔者遇到过下面 3 个问题。

3.4.1 表情在 MySQL 的存储

表情 UTF-8 编码有的是 3 个字节，有的是 4 个字节，所以一般的 UTF 编码（长度只有 3 个字节）是没法存储表情数据的。

在网上看到一个常用的解决方案：把 MySQL 升级到 5.5 以上，然后把字符编码改为 utf8mb4_general_ci。

但实际情况是，有可能在以前的 App 版本中不需要支持表情，这时系统已经运营了一段时间后才升级 MySQL，需要很高的运维成本，同时具备一定的风险，例如，迁移前的不同 MySQL 版本间需要数据同步，保证数据的一致性；迁移过程中可能出现意想不到的事情，造成服务停止。

实践中笔者发现了一个适用于 MySQL 5.1 的表情存储方法：把含有表情数据的字段类型变为 blob，没错，就是用二进制存储，这样就能在改动 MySQL 最少的情况下支持表情数据。

3.4.2 当文字中夹带表情的处理

很多时候如果文字中夹带表情，那么这些文字的处理就会出现问题，例如，如果一个用户的昵称带有表情，那么如何把这个昵称转换为拼音呢？在推送 APNS 过程中，如果推送的文字中夹带表情，推送到 App 端后会显示乱码。

在 App 后台存在着大量要处理文字中夹带表情的需求。笔者遇到了这个问题，先是找到了 https://github.com/iamcal/PHP-emoji 这个转换表情的类库，但发现这个类库不支持 iOS6 后新增的表情，最后没办法了，笔者写了个抓取程序，把 http://punchdrunker.github.io/iosemoji/

table_html/ios6/index.html 中 iOS6 后新增的表情抓取出来，写了个新的类库并开源了 https://github.com/newjueqi/converemojitostr，这个类库的作用就是把文字中夹带的表情替换为一个特殊的字符（默认是“#”）。

3.4.3　Openfire 中发送表情引起连接断开的问题

Openfire 中如果客户端发送某些特殊的字符（例如一些表情符号），xmpp 会断开客户端的连接，经查这个问题是由 Openfire 的以下代码引起的。

```
   src\Java\org\jivesoftware\Openfire\net\MXParser.Java
   protected char more() throws IOException, XMLPullParserException {
   final char codePoint  = super.more(); // note - this does NOT return a
codepoint now, but simply a (single byte) character!
     if ((codePoint == 0x0) ||  // 0x0 is not allowed, but flash clients
insist on sending this as the very first character of a stream. We should
stop allowing this codepoint after the first byte has been parsed.
       (codePoint == 0x9) ||
       (codePoint == 0xA) ||
       (codePoint == 0xD) ||
       ((codePoint >= 0x20) && (codePoint <= 0xD7FF)) ||
       ((codePoint >= 0xE000) && (codePoint <= 0xFFFD)) ||
       ((codePoint >= 0x10000) && (codePoint <= 0x10FFFF))) {
      return codePoint;
     }

     throw new XMLPullParserException("Illegal XML character: " + Integer.
parseInt(codePoint+"", 16));
       }

   在这段代码把特殊的表情符号值当成了一个异常抛出，所以 Openfire 会断开连接。
   解决方法是把特殊的表情符号值也包含，代码修改如下：
       @Override
       protected char more() throws IOException, XMLPullParserException {
    final char codePoint  = super.more(); // note - this does NOT return a
codepoint now, but simply a (single byte) character!
     if ((codePoint == 0x0) ||  // 0x0 is not allowed, but flash clients
insist on sending this as the very first character of a stream. We should
stop allowing this codepoint after the first byte has been parsed.
       (codePoint == 0x9) ||
       (codePoint == 0xA) ||
       (codePoint == 0xD) ||
       //fix some emotion
       ((codePoint >= 0x20) && (codePoint <= 0xFFFD)) ||
```

```
        ((codePoint >= 0x10000) && (codePoint <= 0x10FFFF))) {
      return codePoint;
     }
     throw new XMLPullParserException("Illegal XML character: " + Integer.
parseInt(codePoint+"", 16));
       }
```

3.5 高效更新数据

App 的主页或通知栏经常需要通过 API 获取最新的数据。怎么在这部分上做优化，使获取数据的效率更高呢？本节介绍了推拉结合和数据增量更新这两种实现高效获取数据的策略。

高效更新数据在 App 中的应用场景如图 3-7 所示。

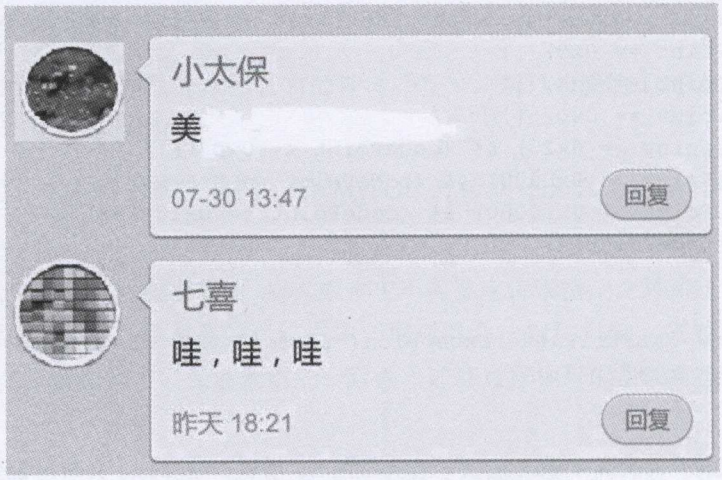


图 3-7　高效更新数据在 App 中的应用场景

如图 3-7，在 App 首页经常会出现瀑布流形式的内容，具体可参考新浪微博的 App。

这种内容有以下的特点。

- 用户访问的频次高（首页一般都经常访问）。
- 数据量大。

那么，怎么才能高效获取这种首页数据呢？

推拉结合和数据增量更新就是实现高效获取数据的关键。

3.5.1　内容的推拉

平常的 App 设计中，如果 App 需要知道首页是否有内容更新，通过一个轮询机制访问获取数据 API，从 API 是否返回更新的数据得知是否有内容更新。

但是轮询的缺点也很明显。

- 耗电。
- 耗流量。

轮询是很典型的拉模式，每隔一段时间 App 向 App 后台发送请求获取数据。这样会耗费大量的网络流量，同时也增大了服务器的压力。

下面的流程图展示了轮询中 App 每隔 5 分钟向 App 后台获取数据的过程，如图 3-8 所示。

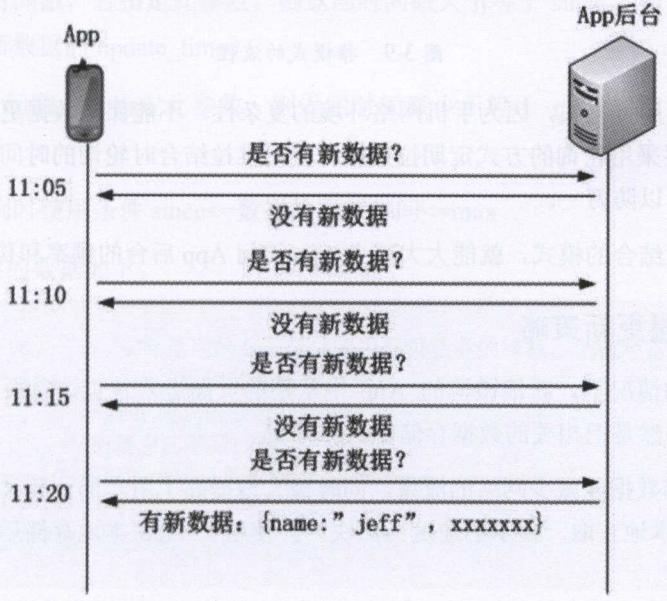


图 3-8　轮询中 App 每隔 5 分钟向 App 后台获取数据

怎么才能减少轮询的次数？答案是通过推模式。每当 App 后台有数据更新，就通过推送系统通知 App，当 App 收到这个数据更新的通知后再调用 API 获取相应的数据。使用推模式的流程如图 3-9 所示。

在推模式中读者可能有疑惑：为什么推送消息给 App 时不把新数据都附上？因为在推送过程中要保持业务的简单性，不把过于复杂的业务整合到推送流程，App 要获取相关的业务数据必须通过调用 App 后台提供的 API。

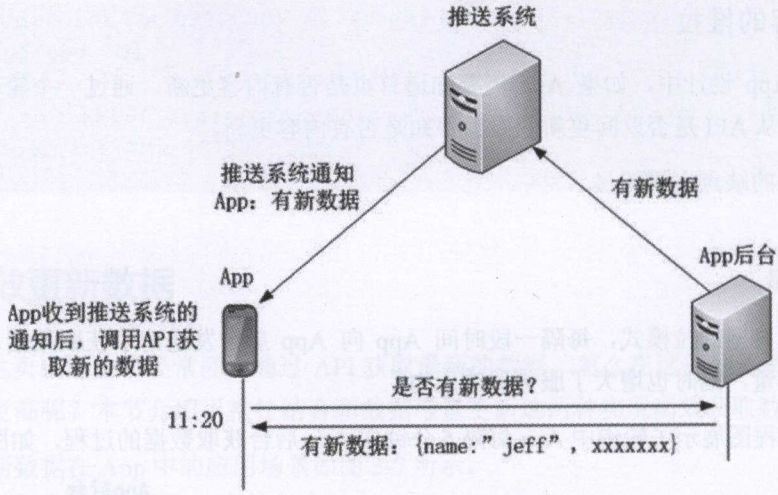


图 3-9　推模式的流程

当然了不能只用推模式，因为手机网络环境的复杂性，不能保证数据更新的通知一定能到达 App，所以也要采用轮询的方式定期拉数据。使用推拉结合时轮询的时间间隔可以设置得比较长，主要是为了以防万一。

通过这种推拉结合的模式，就能大大减少 App 访问 App 后台的频率和传输的数据量。

3.5.2　数据增量更新策略

在没有网络的情况下，新浪微博的 App 中从别的页面进入首页，首页中已经收到的微博还是能显示，这显然是把相关的数据存储在 App 本地。

App 本地存储数据能减少网络的流量，同时极大地提高了用户的体验（读者想一下大量的数据都能在 App 本地获取，显示的速度当然快）。使用了 App 本地存储后，需要考虑的是数据的增量更新方案。

什么是数据的增量更新？假设用户 A 的首页在数据表中是有 40 条数据，id 为 1~40，App 每次通过 API 获取 10 条数据。第一次运行时，App 通过 API 获取了 id 为 1~10 条数据同时存储在 App 本地。假设用户离开了这个页面再回到首页，这时 App 需要再次从数据库中获取数据，由于之前已经有 10 条数据（id 为 1~10）存储在 App 本地了，那么现在需要从数据库中获取的 10 条数据就是从剩余的 30 条数据（id 为 11~40）中获取后并保存在 App 本地。这个就是增量更新的典型例子。

增量更新的原理是在数据库中，每条数据都必须有 update_time 这个值，记录数据最后更新的时间，当 App 从服务器获取了一次数据后（返回的数据必须按时间排序，update_time 最

近的在第一条），记录下所获取的最新数据的 update_time，当再次获取数据就只需要获取 update_time 到访问服务器这刻为止所更新的数据即可。

下面笔者举一个简化的例子详细说明。

首先是一些假设。

1. App 每次请求都带 4 个参数，如 http://test/api/timeline?count=3&page=1&since=1100&max=1200。

   count：每页的显示条数（默认为 3）

   page：当前页码（默认为 1）

   since：时间戳，若指定此参数，则返回时间戳大于等于 since 的结果（应该是上次获取的最新数据的 update_time）

   max：时间戳，若指定此参数，则返回时间戳少于等于 max 的结果（应该是发送时的时间）

   SQL 查询时使用条件 since<=数据的更新时间<=max

2. API 返回的数据如下。

```
{
    "size": 10,    //实际返回的数据量（因为分页获取的缘故，该值经常少于 total 值）
    "total": 284,  //应该返回的总数据量
    "page": 1,     //把请求的参数回传
    "count": 3, //把请求的参数回传
"max": 0, //max 为获取的最后一条数据的 update_time
"since": 0,
"data":{ /*返回的数据实体*/ }
}
```

3. App 存储的本地数据中的 update_time 是指 App 后台中这条数据的更新时间，不是指 App 中这条数据的更新时间。

现在开始讨论。

（1）当 App 安装完毕后还没启动，App 后台的数据表中的数据为 3 条，App 存储的本地数据为空。

App 后台中数据表的数据如表 3-1 所示。

表 3-1　App 后台中数据表的数据

id	update_time
1	1100
2	1101
3	1101

App 存储的本地数据如表 3-2 所示。

表 3-2　App 存储的本地数据

id	update_time

（2）当 App 第一次运行（时间为 11:05），因为是 App 第一次运行，所以 since 为 0，max 为现在的时间点 1105，在 App 后台的数据表中获取所有数据。

发送的请求如下。

```
http://test/api/timeline?count=3&page=1&since=0&max=1105
```

（3）发送请求后 API 返回数据如下。

```
API 返回的数据
{
    "size": 3,    //实际返回的数据量
    "total": 3,   //应该返回的总数据量
    "page": 1,
    "count": 3,
    "max": 1101,
    "since":0,
    "data":{ /*返回的数据实体*/ }
}
```

这时 App 后台数据表中的数据如表 3-3 所示。

表 3-3　App 后台数据表中的数据

id	update_time
1	1100
2	1101
3	1101

App 存储的本地数据如表 3-4 所示。

表 3-4　App 存储的本地数据

id	update_time
1	1100
2	1101
3	1101

这里是重点①：**API 返回数据中的 max 必须为最后一条数据的 update_time**。

（4）现在的时间是 11:20，用户点击了页面中“获取更多”的按钮，App 应该从服务器的数据表中拉取数据，在发送请求前，App 后台的数据表数据如表 3-5 所示。

表 3-5　App 后台的数据表数据

id	update_time
1	1100
2	1101
3	1101
4	1118
5	1118
6	1119
7	1119

从表 3-5 可看到，比起上次拉取数据（查看表 3-1）的时候，App 后台的数据表多了 id 为 4，5，6，7 的数据。

这时发送 API 请求，这里是重点②：**当 API 的返回数据 size=total 时，since 值比上次获取大一点，因为这时数据已经获取完整了，没必要重复获取上次已经获取的数据**（记得条件 since<=update_time<= max 吗？），所以 since 值设置为 1101+1=1102，max 为现在的时间点：1120，请求的 URL 如下。

```
http://test/api/timeline?count=3&page=1&since=1102&max=1120
```

发送请求后 API 返回的数据和 App 存储的本地数据如下。

```
API 返回的数据
{
    "size": 3,    //实际返回的数据量（因为分页获取的缘故，所以经常少于 total 值）
    "total": 4,   //应该返回的总数据量
    "page": 1,
    "count": 3,
    "max": 1119,
    "since":1102,
```

```
    "data":{ /*返回的数据实体*/ }
}
```

App 存储的本地数据如表 3-6 所示。

表 3-6　App 的数据

id	update_time
1	1100
2	1101
3	1101
4	1118
5	1118
6	1119

这里是重点③：**在数据库中，update_time 为 1101~1120 的数据有 4 条，但由于分页的缘故，只获取了 3 条（从 size 和 total 参数可以判定），这意味着 1101~1120 这段时间的数据没有获取完整，App 所获取的最后一条数据的 update_time 是 1119，服务器的数据表中剩下的没有被 App 获取的数据有两种情况：**

- update_time 刚好是 1119。
- update_time 大于 1119。

由于没法判断属于哪种情况，如果下次拉数据的时候 since 大于 1119，服务器的数据表中 id 为 7 的数据将不会获取，那么会造成 App 中丢失了 id 为 7 的数据，所以针对上次数据获取不完整的情况，下次获取数据时 since 必须是等于 1119，虽然有可能会获取重复的数据。

这里是重点④：**当 API 的返回数据 size 少于 total，为了避免有数据丢失，since 为上次收到 API 的返回数据的 max 值：1119，max 为现在的时间点：1120。**关于策略重点④，请结合策略的重点③一起理解。

（5）当 App 端检查到 size 和 total 值不相等，这意味着服务端的数据没有获取完毕，App 端应该再次发送请求获取完整的增量更新数据。

注意：在“获取最近微博”类的瀑布流拉取形式中，其实一般是不需要使用 page 参数的，直接拉取全部最新的数据就好了，笔者之所以要保留这个参数，是为了防止在一页中返回过多数据的情况。

这时 App 端发送 API 请求，请求的 URL 如下。

```
http://test/api/timeline?count=3&page=1&since=1119&max=1120
```

发送请求后 API 的返回数据和 App 存储的本地数据如下。

```
API 返回的数据
{
    "size": 2,    //实际返回的数据量（因为分页获取的缘故，所以经常少于 total 值）
    "total": 2,   //应该返回的总数据量
    "page": 1,
    "count": 3,
    "max": 1119,
    "since":1119,
    "data":{ /*返回的数据实体*/ }
}
```

这里是重点⑤：**API 中返回 id 为 6 的数据已经在 App 的本地数据存在，对于这条数据，App 端应该放弃重复插入。**

最后 App 存储的本地数据如表 3-7 所示。

表 3-7　App 存储的本地数据

id	update_time
1	1100
2	1101
3	1101
4	1118
5	1118
6	1119
7	1119

整个增量更新的策略到这里为止已经分析完毕。

对于这个增量更新的策略，请仔细理解重点①，②，③，④，⑤的分析。

增量更新的策略还要处理一个删除数据的问题。假设 App 后台的数据表要删除一条数据，那怎么通知 App 本地也删除这条数据呢？笔者的解决方案是在服务器的数据表中增加一个标识 is_delete，当需要在业务逻辑上删除的时候，把这条数据的 is_delete 设为 1，同时更新 update_time。当 App 通过增量更新机制获取到这条 is_delete 为 1 的数据，就在 App 本地数据中把这条数据删除。为了避免在 App 后台保存太多的数据，可以在服务器设置一个定时任务（关于定时任务请查看本书“2.7 定时任务”）把那些已经标识 is_delete 设为 1 的数据删除。

上面的例子中依靠修改时间的增量更新策略有个问题：由于更新时间只精确到秒，产生了大量有相同更新时间的数据，造成了③，④，⑤需要处理重复数据的问题，改进方法有两个。

- 使用更少的更新时间，例如毫秒，这样就能减少数据重复的机会。
- 使用数据行的 id 作为更新的标准，由于 id 是永不重复的，更新时只在数据库上获取比上次最后收到的 id 大的数据，这样 App 存储的本地数据就不需要比较数据是否重复。

但是使用了 id 作为更新的标准，在数据库上获取更新的数据也成了一个问题：更新数据只更改数据的其他内容，不会更改数据的 id 值。一个解决方法是把更新数据变成插入一条新数据，在插入的数据中设置属性“updateId”指向需要更新的数据，App 收到这条数据后检查到这个属性就知道其是更新旧数据，如图 3-10 所示。

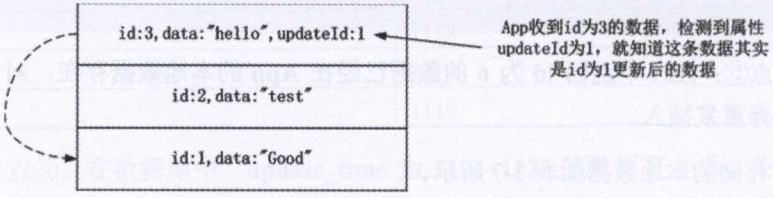


图 3-10　使用 id 的更新策略

删除的策略也是和更新数据类似的策略：插入一条数据，在插入的数据中设置属性“deleteId”指向原来的数据，App 收到这条数据后检查到该属性就知道其是删除旧数据，如图 3-11 所示。

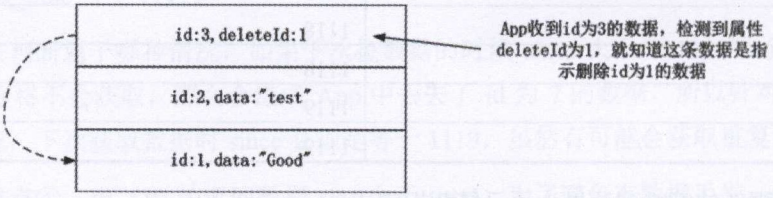


图 3-11　使用 id 的删除策略

3.6　图片处理

App 上线后不断接收用户反馈，有的 App 经过一段时间的反馈和产品调整后会面临 App 改版的问题。

App 改版一般会有比较大的 UI 改动，改动 UI，那么图片的尺寸也必须改变。

在本书“2.2 设计 API 的要点”一节中，笔者提到 App 后台图片处理的一个基本原则：数据库只保存原图的路径。对于同一张图片来说，针对不同机型、不同 App 版本所需要的尺寸不同，使用动态生成的策略，大体思路如下。

（1）App 在图片的 URL 末尾加上参数，其用来声明需要生成的图片的新的尺寸，例如：App 端需要图片（http://www.baidu.com/img/bdlogo.gif）80×80 的尺寸，则在图片的路径加上宽和高的参数（类似于 CDN 的机制） http://www.baidu.com/img/bdlogo.gif? w=80&h=80。

（2）服务器接收到图片的请求，先在缓存中查找这个尺寸的图片是否已经生成，如果已经在缓存中有记录，则不用重新生成。

（3）如果该尺寸的图片还没生成，则生成新的图片尺寸，并把新生成的图片路径放在缓存中，同时把该尺寸图片的路径返回给 App。

App 整体架构中的图片最少有两层缓存。

- App 本地的图片缓存，如果 App 中没有该图片时，才从服务器获取图片。
- 服务器的图片缓存，记录不同尺寸图片的保存路径。

笔者的建议是：直接使用七牛或又拍等文件云存储服务，文件云存储不但可以加速图片的下载/上传，也能实现图片的大量操作（例如图片裁剪、加水印等常用的操作）。图片的上传/下载速度是影响用户体验的一个重要部分！

3.7　视频处理

在热门的社交类 App 中视频到处可见，例如在社交类的 App 上用户可以拍摄属于自己的小视频并发布到相应的栏目中。

App 后台常见的视频处理有以下几种。

- 视频的截图。很多 App 的视频列表上用一张图片表示该视频，这张图片就是通过截取视频的某一帧（通常是第一帧）得到的。
- 出于版权保护的目的给视频加水印。
- 视频转码，允许用户上传手机上的视频，并转换为 App 后台支持的格式。

视频的处理对于大多数程序员来说是个很陌生的领域， 这里介绍一个视频处理最常用的工具：FFmpeg。

3.7.1　FFmpeg 简介

FFmpeg 的官网（http://ffmpeg.org/）是这样介绍的：A complete, cross-platform solution to record, convert and stream audio and video。FFmpeg 可以用来记录、转换数字音频、视频，并能将其转化为流的开源计算机软件。

FFmpeg 是个跨平台的软件，可以在 Linux 下使用，也可以在 Windows 或 Mac 下使用。

这个项目最早由 Fabrice Bellard 发起，现在由 Michael Niedermayer 维护。许多 FFmpeg 的开发人员都来自 MPlayer 项目，而且当前 FFmpeg 项目也是放在 MPlayer 项目组的服务器上。项目的名称来自 MPEG 视频编码标准，前面的“FF”代表“Fast Forward”。

国内的七牛云存储的音视频处理的核心模块也是使用 FFmpeg。

FFmpeg 可以实现的功能有。

- 视频采集。
- 音/视频格式转换。
- 视频抓图。
- 加视频水印。

FFmpeg 主要由以下几个部分组成。

- Libavcodec：包含了所有 FFmpeg 音/视频编解码器的库。为了保证最优性能和高可复用性，大多数编解码器从头开发的。
- Libavformat：包含了所有的普通音/视格式的解析器和产生器的库。

三个实例程序：

- FFmpeg：命令行的视频格式转换程序（一般就直接调用这个文件）。
- FFplay：视频播放程序（需要 SDL 支持）。
- FFserver：多媒体服务器。

3.7.2 后台调用 FFmpeg 的功能

使用 FFmpeg 进行视频转换很简单，例如：把 AVI 转换为 MP4 可使用下面的命令行。

```
ffmpeg -i source.avi -f psp -r 29.97 -b 768k -ar 24000 -ab
64k -s 320×240 destination.mp4
```

在后台语言中怎么调用 FFmpeg 进行格式转换？

一个常见的思路是通过构造命令行的方式，把上面命令行构造出来，然后在后台语言中调用 FFmpeg 执行文件。

有个 Java 调用 FFmpeg 开源项目 jave（主页：http://www.sauronsoftware.it/projects/jave/）就是这样实现。但这个项目中的 FFmpeg 版本已经很旧，如果需要，可以替换 FFmpeg 为最新版本。

下面的例子用 jave 这个项目的代码把某个 AVI 格式的视频转换成 FLV 格式。

```
File source = new File("source.avi");
File target = new File("target.flv");
AudioAttributes audio = new AudioAttributes();
audio.setCodec("libmp3lame");
audio.setBitRate(new Integer(64000));
audio.setChannels(new Integer(1));
audio.setSamplingRate(new Integer(22050));
VideoAttributes video = new VideoAttributes();
video.setCodec("flv");
video.setBitRate(new Integer(160000));
video.setFrameRate(new Integer(15));
video.setSize(new VideoSize(400, 300));
EncodingAttributes attrs = new EncodingAttributes();
attrs.setFormat("flv");
attrs.setAudioAttributes(audio);
attrs.setVideoAttributes(video);
Encoder encoder = new Encoder();
encoder.encode(source, target, attrs);
```

上面代码中通过类 AudioAttributes 和 VideoAttributes 设置了相关的参数，在类 Encoder 中把这些参数构造成命令行执行 FFmpeg 相关的命令。

注意：视频、音频相关的操作一般非常耗费 CPU 资源，在上面的例子中，把视频从 AVI 格式转换成 FLV 格式的过程中，系统的相关资源状况如图 3-12 所示。

```
top - 15:24:50 up  6:06,  3 users,  load average: 0.73, 0.27, 0.19
Tasks: 137 total,   2 running, 135 sleeping,   0 stopped,   0 zombie
Cpu(s): 96.5%us,  2.8%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.7%si,  0.0%st
Mem:   2350560k total,  2274728k used,    75832k free,    11820k buffers
Swap:  3589116k total,    47360k used,  3541756k free,   220124k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
29468 root      20   0 51520  26m 3280 R 89.7  1.2   0:29.58 ffmpeg-amd64
13609 root      20   0  901m 174m  27m S  1.7  7.6   4:56.71 firefox
 1592 root      20   0 53908 1424  820 S  1.3  0.1   0:59.93 VBoxClient
21531 root      20   0 3475m 424m  31m S  1.3 18.5   4:13.82 java
  628 root      20   0  370m 129m 5332 S  1.0  5.6   2:18.64 Xorg
17219 root      20   0 1626m 849m  34m S  1.0 37.0   4:23.24 java
 1650 root      20   0  419m  27m  13m S  0.7  1.2   0:31.28 gnome-panel
13880 root      20   0  270m  22m  12m S  0.7  1.0   0:09.42 gnome-terminal
   18 root      20   0     0    0    0 S  0.3  0.0   0:03.06 kworker/0:1
   20 root      20   0     0    0    0 S  0.3  0.0   0:00.93 kswapd0
  707 nagios    20   0 30428 1080  916 S  0.3  0.0   0:15.03 nagios
  727 root      20   0  106m 2332  996 S  0.3  0.1   0:01.43 php-fpm
  745 root      35  15 18868 3612  956 S  0.3  0.2   0:07.05 preload
 1659 root      20   0  543m  53m  15m S  0.3  2.3   0:24.67 nautilus
29443 root      20   0  775m  21m 9988 S  0.3  0.9   0:00.45 java
    1 root      20   0 24200 1908 1216 S  0.0  0.1   0:02.17 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
```

图 3-12　调用 FFmpeg 时系统的相关资源状况

从上图可看到 FFmpeg 占用了差不多 90%的 CPU 资源。

App 后台要快速处理视频就需要高性能的服务器集群，但是对于创业型公司来说，在服务器这方面的开支是非常不合算的。因此创业型公司可充分运用云服务，例如七牛提供的服务就包含音/视频格式的转换，调用非常方便，如图 3-13 所示。

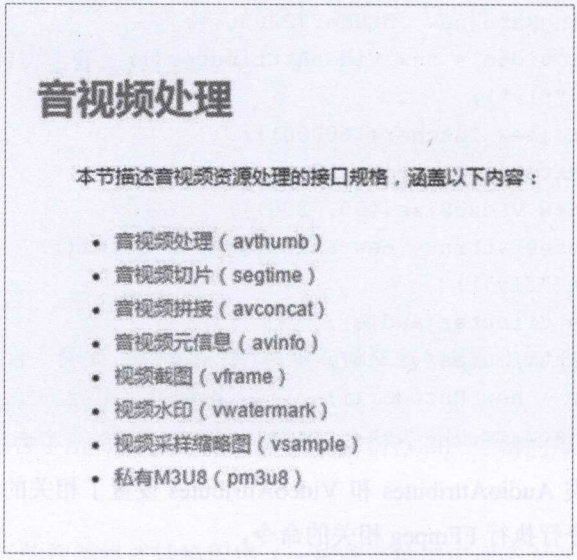


图 3-13　七牛提供的音/视频格式转换

3.8　获取 APK 和 IPA 文件里的资源

移动互联网主要的两个平台是 Android 和 iOS，Android 上文件的安装包是后缀名为 APK 的文件，iOS 上文件的安装包是后缀名为 IPA 的文件，本书分析这两种文件的特点，以及如何在 App 后台解析并获取这些安装包的资源。

3.8.1　Android 的 APK 文件

1. apk 文件的结构

APK 文件其实是 ZIP 格式的压缩包，所以用解压缩软件打开就能打开 APK 文件。

把 APK 文件打开后可看到如图 3-14 所示的文件结构。

assets			文件夹		
lib			文件夹		
META-INF			文件夹		
res			文件夹		
AndroidManife...	7,568	1,896	XML 文档	2013/1/29 10...	AAF003B2
classes.dex	1,110,148	303,979	DEX 文件	2013/1/29 10...	07F95F82
resources.arsc	41,980	41,980	ARSC 文件	2013/1/29 10...	ED2D53...

图 3-14　APK 文件结构

APK 文件的目录结构如下。

- META-INF 目录：存放签名信息，其保证 APK 包的完整性和系统的安全。
- lib 目录：子目录 armeabi 存放 so 文件。
- assets 目录：存放配置文件，这些文件的内容在程序运行过程中可以通过相关的 API 获得。
- res 目录：存放资源文件，包括图片、字符串等等。
- AndroidManifest.XML：该文件是每个应用都必须定义和包含的，其描述了应用的名字、版本、权限、引用的库文件等信息。
- classes.dex：Java 源码编译后生成的 Java 字节码文件（首先是 Java 文件通过 JDK 编译成字节码文件，然后经过 DEX 编译成 classes.dex）。
- resources.arsc：编译后的二进制资源文件的索引（APK 文件的资源表）。

2. 如何获取 APK 文件的基本信息

可以使用 Android-APKtool 获取 APK 文件的基本信息（例如图标、应用名称、版本），下载相应的包后把里面的所有文件复制到/usr/local/bin/ 目录，用如下命令就能获取 APK 的文件信息。

```
/usr/bin/sudo /usr/local/bin/aapt dump badging apk 文件路径
```

执行 Android-APKtool 的例子如图 3-15 所示。

```
[root@192          m]# /usr/bin/sudo /usr/local/bin/aapt dump badging /tmp/test.apk
package: name='com.wws.android' versionCode='9' versionName='1.4'
sdkVersion:'8'
uses-permission:'android.permission.READ_PHONE_STATE'
uses-permission:'android.permission.WRITE_EXTERNAL_STORAGE'
uses-permission:'android.permission.INTERNET'
uses-permission:'android.permission.ACCESS_NETWORK_STATE'
uses-permission:'android.permission.VIBRATE'
uses-permission:'android.permission.READ_LOGS'
uses-permission:'android.permission.ACCESS_WIFI_STATE'
application-label:'手游先锋'
application-icon-120:'res/drawable-ldpi/app_icon.png'
application-icon-160:'res/drawable-mdpi/app_icon.png'
application-icon-240:'res/drawable-hdpi/app_icon.png'
application: label='手游先锋' icon='res/drawable-mdpi/app_icon.png'
launchable-activity: name='com.wws.android.activity.CoverActivity'  label='手游先锋' icon=''
uses-permission:'android.permission.READ_EXTERNAL_STORAGE'
uses-implied-permission:'android.permission.READ_EXTERNAL_STORAGE','requested WRITE_EXTERNAL_STORAGE'
uses-feature:'android.hardware.wifi'
uses-implied-feature:'android.hardware.wifi','requested android.permission.ACCESS_WIFI_STATE, android.permission.CHANGE_WIFI
MULTICAST_STATE permission'
uses-feature:'android.hardware.touchscreen'
uses-implied-feature:'android.hardware.touchscreen','assumed you require a touch screen unless explicitly made optional'
uses-feature:'android.hardware.screen.portrait'
uses-implied-feature:'android.hardware.screen.portrait','one or more activities have specified a portrait orientation'
main
other-activities
other-receivers
other-services
supports-screens: 'small' 'normal' 'large'
supports-any-density: 'true'
```

图 3-15　执行 Android-APKtool 的结果

上面红框中的三行数据分别对应 APK 的版本号、名称、图标。

3.8.2　iOS 的 IPA 文件

1. IPA 文件结构

IPA 文件也是一个 ZIP 文件，用解压缩工具就能解压。IPA 解压后首先出现“payload”文件夹，进入“payload”文件夹后是“应用名.app”文件夹，进入这个文件夹后就是资源的位置。

IPA 文件结构如图 3-16 所示。

IPA 文件的目录结构如下。

- _CodeSignature：文件的签名。
- Info.plist: 加密过的文件，应用名，版本，图标等信息都包含在这个文件中。
- icon2.png，icon.png：不同尺寸的图标文件，也是被加密过的。

_CodeSignature			文件夹		
en.lproj			文件夹		
chat_db_bar@2x.png	3,280	2,981	PNG 文件	2014/7/22 9:30	A814F421
chat_input@2x.png	3,375	3,306	PNG 文件	2014/7/22 9:30	014D18...
DataDemo	1,270,800	432,236	文件	2014/8/13 15...	DC4546...
embedded.mobileprovision	7,964	5,330	MOBILEPROVISI...	2014/8/5 14:33	9B37F881
icon.png	4,628	4,633	PNG 文件	2014/7/31 9:34	15B4A480
icon@2x.png	7,029	6,999	PNG 文件	2014/7/31 9:34	E29CB59E
icon2.png	7,244	7,197	PNG 文件	2014/7/31 9:34	6183FF87
Info.plist	1,418	905	PLIST 文件	2014/8/5 14:33	FF8599A9
LaunchImage-568h@2x.png	14,929	346	PNG 文件	2014/7/31 9:34	5A4C6F74
LaunchImage-700-568h@2x.png	14,929	346	PNG 文件	2014/7/31 9:34	5A4C6F74
PkgInfo	8	8	文件	2014/8/5 14:33	5B8A0449
ResourceRules.plist	150	111	PLIST 文件	2014/7/22 9:30	D1790554

图 3-16　IPA 文件结构

2. 如何获取 IPA 文件的信息

在“1. IPA 文件结构”中提及 IPA 文件的信息是被加密保存在 Info.plist，用开源工具（https://github.com/rodneyrehm/CFPropertyList）就能把里面的内容解密。

简单的用法如下。

```
require_once(__DIR__.'/../classes/CFPropertyList/CFPropertyList.PHP');
$content = file_get_contents("/tmp/Info.plist");
$plist = new CFPropertyList();
$plist->parse($content);
var_dump( $plist->toArray() );
```

$plist 这个数组的信息如下。

```
array(29) {
  'CFBundleName' =>
  string(12) "DataDemo"
  'DTXcode' =>
  string(4) "0511"
  'DTSDKName' =>
  string(11) "iPhoneos7.1"
  'DTSDKBuild' =>
  string(6) "11D167"
  'CFBundleDevelopmentRegion' =>
  string(2) "en"
  'CFBundleVersion' =>    //版本号
  string(3) "2.0"
  .....
' CFBundleDisplayName ' =>    //应用名称
  string(12) "DataDemo"
  .........
```

```
    array(1) {
      'CFBundlePrimaryIcon' =>
      array(1) {
        'CFBundleIconFiles' =>  //图标文件
        array(2) {
          [0] =>
          string(5) "icon2"
          [1] =>
          string(4) "icon"
        }
  }
```

在这个数组中，关键的部分如下。

```
'CFBundleVersion': 版本号
'CFBundleDisplayName': 应用名称
'CFBundlePrimaryIcon'->'CFBundleIconFiles': 图标文件
```

根据这里的图标文件名称，在 IPA 文件中找到被加密的图标文件，使用开源工具（https://github.com/pcans/PngCompote）就能把加密过的图标文件还原。

Pngcompote 的用法。

```
require_once 'pngCompote.php';
$filename = 'Lenna.crush.png'; //需要解密的文件路径
$newFilename = 'Lenna.compote.png'; //解密后的文件路径

$png = new PngFile($filename);
if ($png->revertIPhone($newFilename)) {
        echo 'cleaning done!'.PHP_EOL;
        echo '<img src="'.$newFilename.'"/>'.PHP_EOL;
}
```

3.9 文件系统

现在 App 展现内容的形式多种多样：文字、图片、声音、视频等，其中图片、声音、视频等都是文件，由此可见文件在 App 中占了一个很大的比重。随着 App 不断运营，文件越来越多，占用的磁盘空间也不断增大，设计高效的文件系统对于整个 App 后台架构非常重要。

3.9.1　文件云存储服务

笔者一向推崇创业公司的架构原则是**“尽量使用成熟可靠的云服务和开源软件，自身只专注于业务逻辑”**。

架设文件系统需要牵涉到文件的分布式存储、图片水印、图片缩放，还有 CDN 等方面，每方面都会耗费掉巨大的开发成本和运维成本。

对于中小团队来说，笔者不认为开发人员架设的文件服务比专业的第三方文件云存储服务好，与其在这些基础设施中耗费大量的时间精力，还不如用专业的第三方云存储服务，团队自身只专注于业务逻辑，加快产品的迭代。

而且笔者认为如果计算成本，第三方云存储服务费用比起后端人员开发和运维的成本低多了，想想开发人员钻研和开发文件服务所需的时间就知道。

文件云存储服务的另外一个优势是上传/下载速度非常快，记得笔者第一次使用云存储下载文件时被吓住了，居然达到了 10MB/s 的下载速度，读者想想这么快的下载速度是多美好的用户体验！

3.9.2　架设文件系统

不是每个团队内部都同意使用第三方文件云存储服务，不少老板的想法是必须要掌握核心数据，这时研发人员就只能架设文件系统。

App 后台的文件系统，笔者认为涉及以下 3 个方面。

1. 分布式文件存储系统

对于 App 业务来说，分布式文件存储的基本要求如下。

- 扩容的时候只需要简单地添加机器就能达到扩容的效果，不需要重启整个文件系统上的机器，甚至是迁移文件。
- 保证文件系统高可用、文件冗余备份，避免因某台机器宕机而造成文件服务停止。

移动互联网时代除了视频网盘类的 App 外，大多数 App 以小文件存储为主，所以笔者觉得为解决大文件存储而设计的分片式文件系统不推荐使用，其运维和架构会变得复杂。

笔者推荐的分布式文件存储系统是 FastDFS。

FastDFS 是一个开源的轻量级分布式文件系统，其对文件管理功能包括：文件存储、文件同步、文件访问（文件上传、文件下载）等，解决了大容量存储和负载均衡的问题。 据笔者了解，FastDFS 已经在 UC、56、Kugou 等互联网企业被广泛使用。

FastDFS 的基本原理可以类比生活中的仓库：仓库里面有很多货柜用来存放货物，怎么能

保证仓库里无论增加了多少货柜，货柜都能被合理使用呢？核心是每个仓库里都有一个仓库管理员，仓库管理员知道新增了多少货柜。当工人需要向仓库里放货物时，先问仓库管理员哪个货柜有足够的空间存放货物，仓库管理员在综合考虑货物的大小和各个货柜的剩余空间后，告知工人应该把货物搬到哪个货柜。

FastDFS 就是上面例子中的仓库，FastDFS 里有两大角色：跟踪器（Tracker）和存储节点（Storage）。跟踪器（Tracker）就是仓库管理员，主要做调度工作，在访问上起负载均衡的作用，存储节点（Storage）就是货柜，工人就是向 FastDFS 存储文件的客户端。

存储系统由一个或多个 group 组成，不同的 group 之间文件相互独立，所有 group 的文件容量累加就是整个存储系统中的文件容量。一个 group 可以由一台或多台存储服务器组成，一个 group 下的存储服务器中的文件是相同的，group 中的多个 Storage 起到了冗余备份和负载均衡的作用。

FastDFS 的架构如图 3-17 所示。

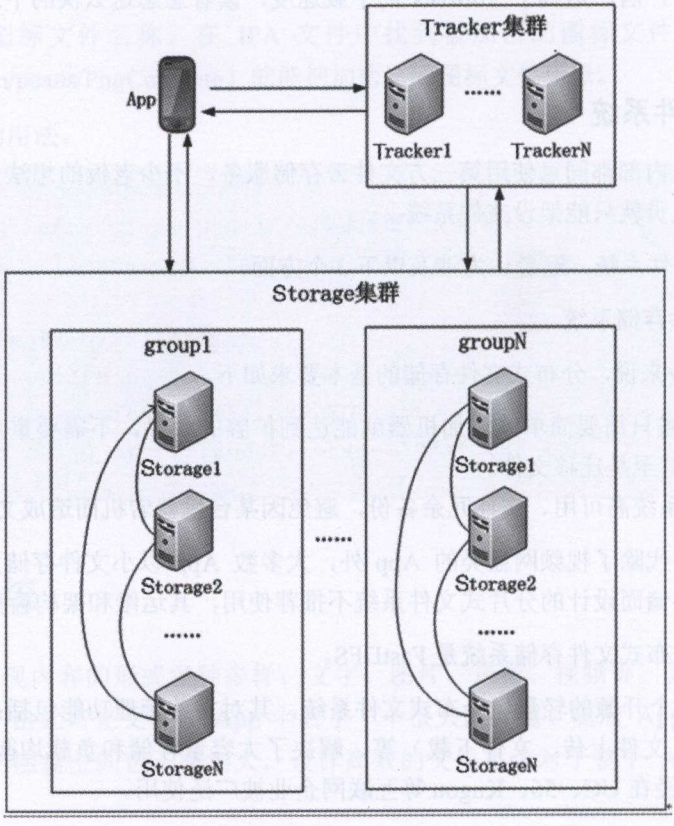


图 3-17　FastDFS 架构图

2. 图片水印，缩放和裁剪

开发人员在 App 后台实现图片的裁切等功能，必须考虑图片操作是非常消耗 CPU、内存等资源和占用大量的磁盘 IO，所以选择图片处理工具要慎重！

笔者推荐使用 GraphicsMagick 作为图片处理软件，其是一个久经考验的软件，支持多个平台，支持多种语言客户端，处理速度快，消耗资源少，并且规模较大的图片网站如 Flickr 都在使用 GraphicsMagick。

3. CDN

CDN 最大的作用是使图片、音频、视频等静态文件下载速度更快，用户体验更好。

App 后台访问量大时通过 CDN，可以把图片、音频、视频等静态文件请求提前响应，不让其到达应用服务器，也是一种应付高并发的方法。

现在除了传统的 CDN 服务商外，阿里云和 UCloud 等服务商也提供了 CDN 服务，同时七牛、又拍等文件云存储服务也具备了 CDN 的功能，上面这些服务都极大地方便开发者。

另外很多 CDN 服务商都提供图片的水印、缩放和裁剪功能，开发者直接使用这些功能就不需要在图片处理上投入开发成本。

3.10　ELK 日志分析平台

读者想象一下下面的场景：有个应用服务器集群里面有 10 台服务器，每台服务器都提供了 API 的接口业务，其使用了负载均衡技术把 API 请求平均分发到每台服务器上，服务器会把处理流程记录在日志中。由于应用服务器集群使用了负载均衡技术，当查找问题时开发人员根本不知道问题出现在哪台服务器，因此开发人员不得不登录所有服务器去逐一查看日志，在这个集群中，开发人员需要分别查看 10 台服务器的日志。如果集群中机器的规模达到 100 台，开发人员需要登录 100 台服务器查看日志，这太麻烦了！

为了解决日志查看不方便和日志中的异常（不能及时报警等问题），后台可以引入 ELK（Logstash+ElasticSearch+Kibana）这个分布式的日志收集和分析系统。

3.10.1　基本模块

这个日志系统各模块功能如下。

- Logstash：收集处理解析日志。其中有两个角色：
  - shipper，在产生日志的机器上运行，发送日志至 indexer；

- indexer：接收并索引化事件。

- ElasticSearch：一个基于 Lucene 的分布式搜索服务，用来提供存储搜索。
- Kibana：一个开源和免费的工具，其可以汇总、分析和搜索重要数据日志并提供友好的 Web 界面，用来报警统计展示。

3.10.2 日志分析流程

ELK 的流程如图 3-18 所示。

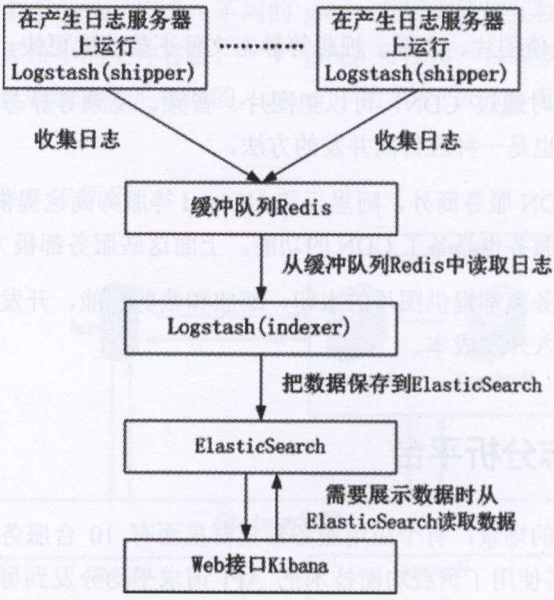


图 3-18 ELK 的流程

ELK 收集和分析日志的流程如下。

（1）在需要收集日志的机器上运行 Logstash（shipper），其用于监控、过滤并收集日志。

（2）Logstash（shipper）把符合条件的日志发送到缓冲队列 Redis。

（3）Logstash（indexer）从缓冲队列 Redis 读取日志，将日志收集在一起交给搜索服务 ElasticSearch。

（4）ElasticSearch 把收集到的日志结合自定义搜索的规则索引起来。

（5）当用户需要在 Kibana 获取数据时，Kibana 向 ElasticSearch 发送请求，在 ElasticSearch 的自定义搜索返回数据的基础上，以友好的页面展示给用户。

3.11　Docker 构建一致的开发环境

在移动互联网企业，开发者完成某个功能需求后上线发布的流程如下。

由开发者在个人电脑上部署新的功能后独自测试，测试完毕后在测试服务器中部署新功能，再由公司内部的测试人员在测试服务器上测试，在测试服务器测试完毕后再在生产服务器部署新功能，正式发布该功能。

这个流程如图 3-19 所示。

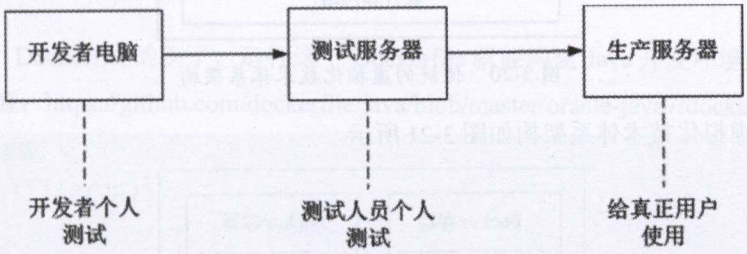


图 3-19　完成功能需求的上线发布的流程

在图 3-19 所示的流程中会产生下面一系列的问题。

- 每台计算机的 Linux 环境、JDK 版本、PHP 版本、Nginx 版本、MySQL 版本等不一致，有可能造成某些问题只是某台计算机上出现，其他电脑没法重现。
- 某些开发环境搭建复杂，还要分别在个人计算机、测试服务器、生产服务器上搭建一次，里面有大量的重复劳动，效率低下。
- 在个人计算机和测试服务器上有可能只是单机环境，到了生产服务器上变成了分布式环境，环境不一致导致部署方法不一致，造成在测试阶段没法发现问题。

Docker 是一个用于统一开发和部署的轻量级容器，让开发者打包其应用及相关的依赖包到一个可移植的容器，发布该容器到其他机器，就能很容易地实现应用的部署。

3.11.1　Docker 原理

传统的虚拟化技术体系在服务器操作系统上安装了多个虚拟机，每个虚拟机上通过虚拟化技术实现了一个虚拟操作系统，在这个虚拟操作系统上运行应用。传统的虚拟化技术体系架构如图 3-20 所示。

Docker 的虚拟化技术体系在服务器的操作系统上有一个 Docker 服务在运行，在这个 Docker 服务上运行着多个 Docker 容器，每个 Docker 容器中运行着应用，容器与容器间的应用是相互隔离、相互独立的，但通过 Docker 服务占用着服务器的硬件和网络资源。

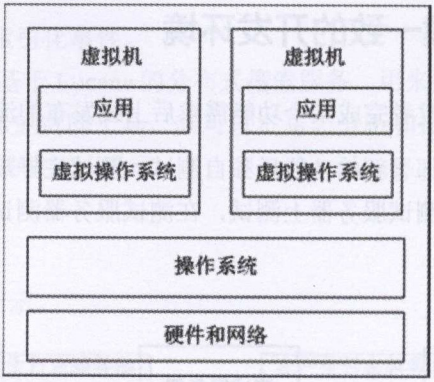


图 3-20　传统的虚拟化技术体系架构

Docker 的虚拟化技术体系架构如图 3-21 所示。

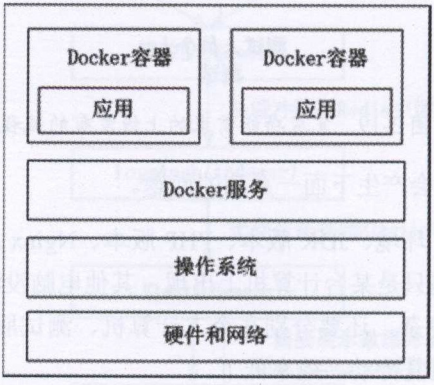


图 3-21　Docker 的虚拟化技术体系架构

3.11.2　搭建一致的开发环境

Docker 中有 3 个关键概念。

- 镜像（Image）：Docker 镜像（Image）类似于虚拟机镜像，可以把其理解为 Docker 的只读模板，其包含了文件系统。
- 容器（Container）：Docker 容器（Container）类似于一个沙箱，Docker 使用容器隔离资源，并在其内部运行应用，可看成是一个简易版的 Linux 环境。镜像是只读，容器从镜像启动后，Docker 会在镜像最上层创建一个可写层，这样镜像本身就能保持不变。
- 仓库（Repository）：Docker 仓库（Repository）类似于代码仓库，是 Docker 集中存放镜像文件的场所。

Docker 仓库里面包含了大量操作系统的基础镜像（例如 CentOS、Ubuntu 等），开发者从

仓库中拉取这些操作系统的基础镜像后就能在其基础之上构建自身环境的镜像。

Docker 提供了 Dockerfile 这种脚本给开发者创建自定义的镜像。开发者通过 Dockerfile，很容易在操作系统的基础镜像（例如 CentOS、Ubuntu 等）上安装指定的软件及其依赖，从而构建一个适用于自身业务环境的镜像。

简单来说，读者可以把操作系统的基础镜像理解为一个干净版操作系统，通过 Dockerfile（相当于 Linux 上的安装脚本）往这个干净版操作系统上安装需要的软件后，再生成一个新的镜像。

下面是一个 Dockerfile 的例子，运行这个 Dockerfile 就能构建 Java 开发环境的 Docker 镜像（Dockerfile 来源：https://github.com/dockerfile/java/blob/master/oracle-java7/dockerfile）。

```
# 下载基础镜像
FROM dockerfile/ubuntu

# 安装 Java.
Run echo oracle-Java7-installer shared/accepted-oracle-license-v1-1 select
true | debconf-set-selections && \
  add-apt-Repository -y ppa:Webupd8team/Java && \
  apt-get update && \
  apt-get install -y oracle-Java7-installer && \
  rm -rf /var/lib/apt/lists/* && \
  rm -rf /var/cache/oracle-jdk7-installer

# 定义工作目录
WORKDIR /data

# 定义 JAVA_HOME 环境变量
ENV JAVA_HOME /usr/lib/jvm/java-7-oracle

# 定义默认的命令
CMD ["bash"]
```

使用 Docker 构建一致的开发环境是依赖于 Dockerfile：把编写完成的 Dockerfile 放置在版本管理服务器中，在不同的服务器上获取这个 Dockerfile 并运行就能构建相同的镜像，从而得到一致的开发环境。这个流程如图 3-22 所示。

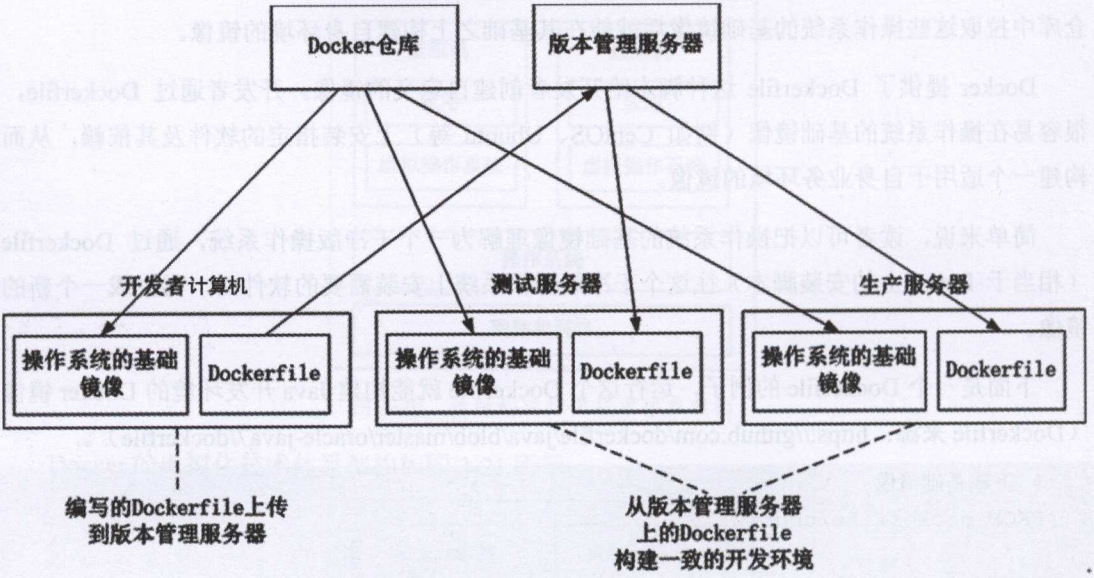


图 3-22　用 Docker 构建开发环境流程

第 4 章
Linux——App 后台应用最广泛的系统

App 后台开发人员需要兼顾开发与运维两方面。现在的 App 后台服务器大多数是运行在 Linux 系统，因此开发工作中无可避免地涉及大量 Linux 的运维操作。

本章主要介绍 Linux 下面的内容。

- 基本的系统优化。
- 常用的运维命令。
- 故障分析案例。

4.1 基本的系统优化

App 后台的 Linux 系统如果是采用默认安装或者机房的工作人员帮忙安装，运维人员需要对其进行优化，以获得更高的性能和更大的安全性。

注：为了演示方便，本书的 Linux 相关操作将会以 root 的权限演示。

4.1.1 开机自启动服务优化

Linux 启动时会首先启动一个称为 init 的进程，然后由其来启动后面的任务，包括多用户环境、网络等。运行级是操作系统当前运行的功能级别，这个级别从 1 到 6，具有不同的功能，这些级别在/etc/inittab 文件里指定，这个文件是 init 进程寻找的主要文件。

可在/etc/inittab 上看到描述。

```
# Default runlevel. The runlevels used by RHS are:
#   0 - halt (Do NOT set initdefault to this) ，表示关机
#   1 - Single user mode，单用户模式
#   2 - Multiuser, without NFS (The same as 3, if you do not have networking)，
无网络连接的多用户命令行模式
#   3 - Full multiuser mode，有网络连接的多用户命令行模式
#   4 - unused，不用
#   5 - X11，带图形界面的多用户模式
#   6 - reboot (Do NOT set initdefault to this)，重新启动
```

可用 runlevel 查看当前的运行级别（注意，这个命令只能在 root 下运行）。

chkconfig 命令主要用来更新和查询系统服务的运行级信息。当需要查询当前系统级服务的运行信息时，可采用下面的命令。

语法：

```
chkconfig [--add][--del][--list][系统服务]
```

参数用法：

--add：添加系统服务。

--del：删除系统服务。

--list：显示所有运行级系统服务的运行状态信息。

例如，显示系统中运行在 3 级别的服务。

```
[root@mode ~]# chkconfig --list|grep 3:on
crond           0:off   1:off   2:on    3:on    4:on    5:on    6:off
network         0:off   1:off   2:on    3:on    4:on    5:on    6:off
sshd            0:off   1:off   2:on    3:on    4:on    5:on    6:off
syslog          0:off   1:off   2:on    3:on    4:on    5:on    6:off
```

当需要更新当前系统服务的运行级别时，可采用下面的命令。

语法：

```
chkconfig [--level <等级代号>][系统服务][on/off/reset]
```

参数用法：

--level：服务的等级。

--on：开启系统服务。

--off：关闭系统服务。

--reset：重置系统服务。

下面以添加 Nginx 为系统服务为例子演示如何添加一个系统服务。

（1）把 Nginx 的启动脚本放在/etc/ini.d/目录下，完整的路径为/etc/ini.d/nginx。

（2）在 Nginx 加入系统服务：

```
chkconfig --add nginx
```

（3）修改 Nginx 服务的运行级别：

```
chkconfig --level 35 nginx on
```

自启动的服务必须遵从最少化的原则，即在保证系统正常的情况下，不需要的服务不启动。

4.1.2　增大文件描述符

当进程打开现有文件或创建新文件时，内核向进程返回一个文件描述符。对 Linux 内核来说，所有打开的文件都通过文件描述符引用。文件描述符是一个非负整数。当进程读或写一个文件时，进程使用 open 或 creat 函数返回文件描述符，标识该文件，将其作为参数传递给 read 或 write 函数后继续读写。

Linux 系统中经常出现的错误“Too many open files”就是由于打开的文件数超过了文件描述符的限制导致。

查询系统文件描述符大小的命令如下。

```
[root@modecron]# ulimit -n
1024
```

修改系统文件描述符的限制有两种方法。

方法一，只对当前的 session 有效的修改方法。

```
[root@modecron]# ulimit -HSn 65535
[root@modecron]# ulimit -n
65535
```

方法二，修改配置文件使永久生效。

```
在/etc/security/limits.conf 中加入
*    -     nofile    65535
```

内核参数对文件描述符也有限制，如果设置的值大于内核的限制也是不行。查看内核参数中文件描述符的值可用如下的命令。

```
[root@mode ~]# sysctl -a|grep file-max
fs.file-max = 1024
```

修改内核参数中文件描述符的值为 65535 可用如下的命令：

```
sysctl -w fs.file-max=65535
```

实际开发过程中，如果在系统低负载的情况下出现错误“Too many open files”主要的原因是所开发程序的问题。由于程序中出现 Bug，导致了打开了大量的文件连接（网络连接也会占用文件描述符）没有及时释放，超过了文件描述的大小而导致出现错误“Too many open files”。程序申请的资源在用完后必须及时释放，这才是从根本上解决错误“Too many open files”的方法。

4.2 常用的命令

本章介绍了笔者日时在系统管理，故障排除和系统调优时的常用 Linux 命令，通过详细了解这些命令，读者可对整个 Linux 系统的状况有更深入的了解。

4.2.1 全面了解系统资源情况——top

top 命令是 Linux 下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况。如果在前台执行该命令，其将独占前台，直到用户终止该程序为止。

top 命令提供实时的系统处理器的状态监视，其将显示系统中 CPU 最“敏感”的任务列表。该命令可以按 CPU 使用率、内存使用情况和进程执行时间对任务进行排序；而且该命令的很多特性都可以通过交互式命令或者在个人定制文件中进行设定。

例如做压测的时候使用 top 命令，可以动态地显示系统的资源、当前系统的内存、CPU 整体使用等情况，某个进程的 CPU、内存使用等情况，让开发者对这些情况有所了解。

在终端输入 top 命令后，如图 4-1 所示。

```
top - 19:10:10 up  9:10,  1 user,  load average: 0.00, 0.00, 0.00
Tasks:  71 total,   1 running,  70 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1018600k total,   943932k used,    74668k free,    30460k buffers
Swap:        0k total,        0k used,        0k free,   728328k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0 19232 1140  848 S  0.0  0.1   0:00.58 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.12 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    6 root      RT   0     0    0    0 S  0.0  0.0   0:00.04 watchdog/0
    7 root      20   0     0    0    0 S  0.0  0.0   0:01.46 events/0
    8 root      20   0     0    0    0 S  0.0  0.0   0:00.00 cgroup
    9 root      20   0     0    0    0 S  0.0  0.0   0:00.00 khelper
```

图 4-1　top 命令

top 命令作为常用性能分析命令，后台开发人员务必要了解输出结果中每一项的具体含义。

top 命令显示的前 5 行是统计信息，第一行是基本信息。

信息栏	含义
19:10:00 up 9:10	当前时间和系统运行时间，格式为时:分。这里表示已经运行了 9 小时 10 分
1 user	当前登录用户数
load average：0.00 0.00 0.00	系统负载。三个数值分别为 1 分钟、5 分钟、15 分钟前到现在的平均值

第二行是任务的信息。

信息栏	含义
total	进程总数
running	正在运行的进程数
sleeping	睡眠的进程数
stopped	停止的进程数
zmobie	僵尸进程数

笔者在这里解析一下僵尸进程，对于僵尸进程这个概念可能很多读者都不太了解。僵尸进程指的是那些已经终止，但仍然保留一些信息的进程，等待父进程为其调用 wait()系统调用来获取子进程的退出状态和其他的信息（即为其扫尾）。当调用父进程为其调用 wait()后，僵尸进程就完全从内存中移除。

僵尸进程无法使用 kill 清理。如果开发者需要手动清理僵尸进程，要找到其父进程，把父进程 kill 掉后 Linux 的 init 将接管其子进程。Linux 中任何一个子进程都必须要有父进程，当父进程被 kill 后，其所有子进程过继给 Linux 的 init 进程，init 进程成为僵尸进程的新父进程，init 进程隔一段时间去调用 wait() 系统调用来清除僵尸进程。

第三行是 CPU 利用率的统计信息。

信息栏	含义
us	User Time，CPU 执行用户进程百分比，包括 NiceTime
sy	System Time，CPU 在内核运行百分比，包括 IRQ 和 SoftIRQ 百分比
ni	Nice Time，调整进程优先级所用百分比
id	Idle Time，系统空闲百分比
wa	Waiting Time，CPU 等待 I/O 完成所用百分比

续表

信息栏	含义
hi	Hard IRQ Time，硬中断占用的 CPU 时间百分比
si	Soft IRQ Time，软中断占用的 CPU 时间百分比
st	Steal Time，虚拟服务占用的 CPU 时间百分比

第四行是内存的使用信息。

信息栏	含义
total	总物理内存
used	已使用的物理内存
free	空闲的物理内容
buffers	缓冲的总量

第五行是交换区的使用信息。

信息栏	含义
total	交换区的总大小
used	已使用交换区的总大小
free	未使用交换区的总大小
cached	缓存的总量。这项信息应该属于上一行的内存使用信息，估计是为了美观，就把这项信息显示在这行

关于第四、第五行的内存和交换区的信息，下面两点需要注意。

1. 什么是交换区（Swap）

Linux 的交换区（Swap）：交换区是硬盘上的一块空间。在内存不足的情况下，操作系统先把内存中暂时不用的数据存到硬盘的交换区，腾出内存来让别的程序运行。

阿里云服务器上的 Linux 系统默认是没有设置 Swap。由于开启 Swap 分区会导致硬盘 IO 性能下降，因此阿里云服务器初始状态未配置 Swap，如果某些应用需要开启 Swap 分区，也可通过相应的命令开启。

如图 4-2 所示是 UCloud 上 top 命令的截图，由图上可获知 UCloud 上的 Linux 系统是设置了 Swap 分区。

Linux 系统在内存不足的情况下会使用 Swap 分区，这时整个系统的性能急降。如果使用 top 命令观察到 Swap 上 used 值不用为 0，那运维人员需要小心了，used 值不用为 0 意味着内存已经不足。这时运维人员先排查有没有不正常使用内存的程序，如果没找到的程序就意味内存已经不足以应对业务的发展，需要升级内存。

```
Tasks: 105 total,   1 running, 104 sleeping,   0 stopped,   0 zombie
Cpu(s):  3.7%us,  4.5%sy,  0.0%ni, 91.1%id,  0.0%wa,  0.0%hi,  0.6%si,  0.1%st
Mem:   3858412k total,  3227920k used,   630492k free,   188300k buffers
Swap:   524280k total,     6912k used,   517368k free,  2301468k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1442 root      20   0  107m 7960 3976 S  4.0  0.2   6528:24 php
 1463 root      20   0  156m  56m 4324 S  4.0  1.5   6511:38 php
```

图 4-2　UCloud 上 top 命令的截图

2. cached 和 buffers 区别

cached 和 buffers 都是内存中存放的数据，简单来说，cached 是存放从磁盘中读取出的数据，buffers 是存放准备写入磁盘的数据。

cached 是 Linux 把文件从硬盘中读取后在内存中保存的数据，下次要读取这些数据时若在 cached 命中（找到数据），则不去读硬盘的数据，否则读硬盘的数据。cached 中的数据按照读取的频率组织起来，最频繁读取的数据放在最快读取到的地方，把不再读取的数据往后排，直到因为 cached 空间不足而删除。如果其他应用面临内存不足的情形，Linux 会把 cached 中的文件清理，多腾出给其他应用使用。

cached 实际缓存的是块而不是文件，块在 Linux 中是磁盘 I/O 操作的最少单位（在 Linux 中，默认是 1KB），这时 cached 就能保存文件、设备和其他非文件系统的数据。

Linux 中的缓存是没有固定大小的，Linux 自动使用空闲内存作为 cached，当内存空间变小时，为了避免内存空间不足而使用 Swap，Linux 会自动释放部分 cached 的内存空间。

buffers 是为磁盘的读写设计的，把分散的磁盘操作集中起来，减少了磁盘寻道的时间和磁盘碎片，从而提高系统性能。在 Linux 中有个守护进程会定期把 buffers 中的数据写入到磁盘，也可以通过命令 sync 手动把 buffers 数据刷到磁盘中。

cached 和 buffers 都是由操作系统管理，只要 Swap 分区没有被使用，即使 cached 和 buffers 占用了很多内存，运维人员都不需要太担心。

运维人员可以使用如下的命令手动清理 cached 中的数据。

```
sync&& echo3 >/proc/sys/vm/drop_caches
```

继续返回 top 命令的讲解，下面这行是进程统计信息区，显示了各个进程的详细信息，如图 4-3 所示。

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0 19232 1140  848 S  0.0  0.1   0:00.58 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
```

图 4-3　进程统计信息区

上面各列的含义如下。

信息栏	含义
PID	进程 id
USER	进程所有者的用户名
PR	优先级
NI	nice 值。负值表示高优先级，正值表示低优先级
VIRI	进程使用的虚拟内存总量，单位 KB
RES	进程使用的、未被换出的物理内存大小，单位 KB
SHR	共享内存大小，单位 KB
S	进程状态（D=不可中断的睡眠状态，R=运行，S=睡眠，T=跟踪/停止，Z=僵尸进程）
%CPU	上次更新到现在的 CPU 时间占用百分比
%MEM	进程使用的物理内存百分比
TIME+	进程使用的 CPU 时间总计，单位 1/100 秒
COMMAND	命令名/命令行

这一栏中常用的列有 5 个：PID（进程的 ID），USER（进程所有者的用户名），%CPU（上次更新到现在的 CPU 时间占用百分比），%MEM（进程使用的物理内存百分比），COMMAND（命令名/命令行）。

注意 top 命令是有交互操作，例如，在 top 命令中按下数字“1”，可以显示 CPU 每个核的使用情况，如图 4-4 所示的下画线。

```
top - 23:09:11 up 13:09,  1 user,  load average: 0.00, 0.00, 0.00
Tasks:   1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1018600k total,   944756k used,    73844k free,    31084k buffers
Swap:        0k total,        0k used,        0k free,   728416k cached
```

图 4-4　显示 CPU 的核运行情况

在 top 命令中输入“h”，可看到 top 命令的详细帮助，如图 4-5 所示。

```
Help for Interactive Commands - procps version 3.2.8
Window 1:Def: Cumulative mode Off.  System: Delay 3.0 secs; Secure mode Off.

  Z,B       Global: 'Z' change color mappings; 'B' disable/enable bold
  l,t,m     Toggle Summaries: 'l' load avg; 't' task/cpu stats; 'm' mem info
  1,I       Toggle SMP view: '1' single/separate states; 'I' Irix/Solaris mode

  f,o     . Fields/Columns: 'f' add or remove; 'o' change display order
  F or O  . Select sort field
  <,>     . Move sort field: '<' next col left; '>' next col right
  R,H     . Toggle: 'R' normal/reverse sort; 'H' show threads
```

图 4-5　top 命令的帮助

由于 top 命令是动态显示全部进程的使用情况，如果只需要观察某个进程的资源使用情况，

用如下的命令。

```
top -p 进程 id
```

如图 4-6 所示是用 top 命令显示 MySQL 进程（进程 id：7967）的资源使用情况。

```
top - 23:10:14 up 13:10,  1 user,  load average: 0.00, 0.00, 0.00
Tasks:   1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.3%us,  0.0%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1018600k total,   944756k used,    73844k free,    31084k buffers
Swap:        0k total,        0k used,        0k free,   728416k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 7967 mysql     20   0  305m  36m 3988 S  0.0  3.6   0:10.22 mysqld
```

图 4-6　用 top 命令 MySQL 进程的资源使用情况

4.2.2　显示进程状态——ps

ps 命令是显示当前系统中进程的状态，这个命令显示的只是运行 ps 命令瞬间的状态，如果需要显示不断更新的状态，请使用上节提及的“top –p 进程 id”命令。

ps 命令的常用参数如下。

a：按用户名和启动时间的顺序来显示进程。

u：显示所有用户的所有进程（包括其他用户）。

x：显示无控制终端的进程。

ps -aux 命令运行后如图 4-7 所示。

```
[root@jeff ~]# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.1  19232  1140 ?        Ss   Jul10   0:00 /sbin/init
root         2  0.0  0.0      0     0 ?        S    Jul10   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    Jul10   0:00 [migration/0]
root         4  0.0  0.0      0     0 ?        S    Jul10   0:23 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S    Jul10   0:00 [migration/0]
root         6  0.0  0.0      0     0 ?        S    Jul10   0:11 [watchdog/0]
root         7  0.0  0.0      0     0 ?        S    Jul10   5:35 [events/0]
root         8  0.0  0.0      0     0 ?        S    Jul10   0:00 [cgroup]
root         9  0.0  0.0      0     0 ?        S    Jul10   0:00 [khelper]
root        10  0.0  0.0      0     0 ?        S    Jul10   0:00 [netns]
root        11  0.0  0.0      0     0 ?        S    Jul10   0:00 [async/mgr]
root        12  0.0  0.0      0     0 ?        S    Jul10   0:00 [pm]
root        13  0.0  0.0      0     0 ?        S    Jul10   0:00 [xenwatch]
root        14  0.0  0.0      0     0 ?        S    Jul10   0:00 [xenbus]
root        15  0.0  0.0      0     0 ?        S    Jul10   0:28 [sync_supers]
root        16  0.0  0.0      0     0 ?        S    Jul10   0:28 [bdi-default]
root        17  0.0  0.0      0     0 ?        S    Jul10   0:00 [kintegrityd/0]
root        18  0.0  0.0      0     0 ?        S    Jul10   0:02 [kblockd/0]
root        19  0.0  0.0      0     0 ?        S    Jul10   0:00 [kacpid]
root        20  0.0  0.0      0     0 ?        S    Jul10   0:00 [kacpi_notify]
root        21  0.0  0.0      0     0 ?        S    Jul10   0:00 [kacpi_hotplug]
```

图 4-7　top 命令

从图 4-7 中可以看到显示的是所有的进程，如果只显示某个进程，可使用如下的命令。

```
ps axu|grep 进程名
```

这里使用了 Linux 的管道命令“|”，其能把前一个命令的输出信息（STDOUT）作为 STDIN 传递给下一个命令，原理如图 4-8 所示。

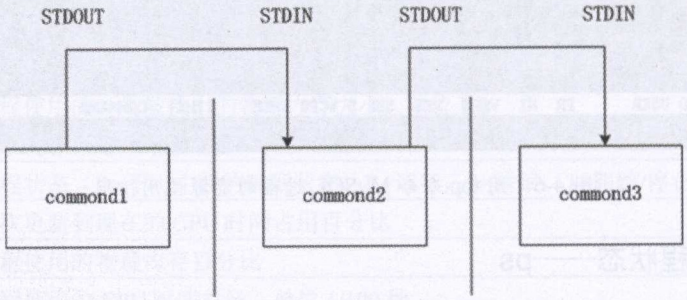


图 4-8　管道命令“|”原理

grep 命令是一种强大的文本搜索工具，其能使用正则表达式搜索文本，并把匹配的行输出。

通过 ps，管道命令“|”和 grep 就能把输出中匹配进程名的行检索出来。

例如需要了解当前系统中 PHP 进程的状态，可使用如下的命令。

```
ps axu|grep php
```

结果如图 4-9 所示：

```
[root@jeff ~]# ps axu|grep php
root      7982  0.0  0.5 206336  5496 ?        Ss   Jul10   0:02 php-fpm: master process (/usr/local/php/etc/php-fpm.conf)
www       7984  0.0  0.5 206336  5196 ?        S    Jul10   0:00 php-fpm: pool www
www       7985  0.0  0.5 206336  5200 ?        S    Jul10   0:00 php-fpm: pool www
root      8368  0.0  0.0 103176   868 pts/0    S+   18:10   0:00 grep php
```

图 4-9　用 ps 命令显示 PHP 进程的状态

在 ps 命令中没找到 PHP 的进程就意味着 PHP 进程已经不存在，这时运维人员需要通过 PHP 的错误日志查找 PHP 进程为什么会不存在，是没启动还是闪退，或者是什么原因导致。

4.2.3　查看网络相关信息——netstat

在日常服务器运维中经常需要查看服务器的网络连接情况，比较典型的需求如下。

- 查看某个端口是否开启。
- 查看某个端口是由哪个程序开启。
- 查看某个端口的连接数。

netstat 命令就可以解决上面的需求。

netstat 命令的常用参数如下。

```
-l: listen，监听的端口。
-a: 显示所有的 Socket，包括正在监听。
-n: 显示数字格式的地址。
-t: 监听 TCP 的端口。
-u: 监听 UDP 的端口。
-p: 显示建立相关链接的程序名。
```

例如，查看系统中启动的端口，可用如下命令。

```
netstat-lntup
```

命令结果如图 4-10 所示。

```
[root@jeff ~]# netstat -lntup
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address               Foreign Address             State       PID/Program name
tcp        0      0 0.0.0.0:80                  0.0.0.0:*                   LISTEN      7989/nginx
tcp        0      0 0.0.0.0:22                  0.0.0.0:*                   LISTEN      878/sshd
tcp        0      0 0.0.0.0:3306                0.0.0.0:*                   LISTEN      7967/mysqld
udp        0      0 123.57.236.144:123          0.0.0.0:*                               1183/ntpd
udp        0      0 10.51.63.41:123             0.0.0.0:*                               1183/ntpd
udp        0      0 127.0.0.1:123               0.0.0.0:*                               1183/ntpd
udp        0      0 0.0.0.0:123                 0.0.0.0:*                               1183/ntpd
```

图 4-10　netstat 查看中启动的端口

查看某个端口是否有启动，可以直接在上面的结果中查看，如果输出的结果很多，需要使用管道命令“|”和 grep 把输出中匹配进程名的行检索出来。

例如，要查看系统中是否有开启 80 端口，使用如下命令。

```
netstat-lntup|grep 80
```

结果如图 4-11 所示，读者从图中看到，Nginx 进程开启了 80 端口。

```
[root@jeff ~]# netstat -lntup|grep 80
tcp        0      0 0.0.0.0:80                  0.0.0.0:*                   LISTEN      7989/nginx
```

图 4-11　netstat 查看系统是否有开启 80 端口

如果要查看系统中 80 端口的连接情况，可用如下所示命令。

```
netstat-nat|grep 80
```

结果如图 4-12 所示。

```
[root@jeff ~]# netstat -nat|grep 80
tcp        0      0 0.0.0.0:80                  0.0.0.0:*                   LISTEN
tcp        0      0 123.57.236.144:43740        140.205.140.205:80          ESTABLISHED
```

图 4-12　netstat 查看系统中 80 端口的连接情况

4.2.4 查看某个进程打开的所有文件——lsof

查看某个 App 的后台错误日志时，如果出现错误“Too many open files”，前面提到是由于打开的文件数超过了文件描述符的限制导致的。如果需要确定某个应用到底打开了多少文件，就需要用到 lsof 命令。

lsof（list open files）是一个列出当前系统打开文件的工具。

lsof 常用的参数如下。

```
-p: 进程的 id
```

通过 top 命令或者 ps 命令，就能查到某个进程 id。

如果在错误日志中看到有很多“Too many open files”，但不知道具体是打开了哪些文件导致这个错误（有时候看到哪些文件被大量打开就能大概知道是程序哪段代码出了问题），lsof 就能派上用场。

例如查看 Nginx 打开了哪些文件，在 ps 命令中查到 Nginx 的进程 id 为 7989，则可用如下的命令。

```
lsof -p 7989
```

结果如图 4-13 所示。

```
[root@jeff ~]# lsof -p 7989
COMMAND  PID USER   FD   TYPE             DEVICE SIZE/OFF    NODE NAME
nginx   7989 root  cwd    DIR              202,1     4096 1188178 /root/lnmp1.1-full
nginx   7989 root  rtd    DIR              202,1     4096       2 /
nginx   7989 root  txt    REG              202,1  5310861 1077220 /usr/local/nginx/sbin/nginx
nginx   7989 root  mem    REG              202,1    65928  262213 /lib64/libnss_files-2.12.so
nginx   7989 root  mem    REG              202,1   122040  265466 /lib64/libselinux.so.1
nginx   7989 root  mem    REG              202,1   110960  269090 /lib64/libresolv-2.12.so
nginx   7989 root  mem    REG              202,1    10192  269134 /lib64/libkeyutils.so.1.3
nginx   7989 root  mem    REG              202,1    43728  270885 /lib64/libkrb5support.so.0.1
nginx   7989 root  mem    REG              202,1   174840  269139 /lib64/libk5crypto.so.3.1
nginx   7989 root  mem    REG              202,1    14664  270881 /lib64/libcom_err.so.2.1
nginx   7989 root  mem    REG              202,1   941920  270884 /lib64/libkrb5.so.3.3
nginx   7989 root  mem    REG              202,1   277704  268696 /lib64/libgssapi_krb5.so.2.2
nginx   7989 root  mem    REG              202,1   469528  262150 /lib64/libfreebl3.so
nginx   7989 root  mem    REG              202,1  1921176  262157 /lib64/libc-2.12.so
nginx   7989 root  mem    REG              202,1    88600  262209 /lib64/libz.so.1.2.3
nginx   7989 root  mem    REG              202,1    19536  262173 /lib64/libdl-2.12.so
nginx   7989 root  mem    REG              202,1  1963296 1053217 /usr/lib64/libcrypto.so.1.0.1e
nginx   7989 root  mem    REG              202,1   441256 1053219 /usr/lib64/libssl.so.1.0.1e
nginx   7989 root  mem    REG              202,1   118342 1077099 /usr/local/lib/libpcre.so.0.0.1
nginx   7989 root  mem    REG              202,1    40400  262161 /lib64/libcrypt-2.12.so
nginx   7989 root  mem    REG              202,1   142640  262181 /lib64/libpthread-2.12.so
nginx   7989 root  mem    REG              202,1   154624  262151 /lib64/ld-2.12.so
nginx   7989 root  DEL    REG                0,4          194985 /dev/zero
nginx   7989 root    0u   CHR                1,3      0t0   3866 /dev/null
nginx   7989 root    1u   CHR                1,3      0t0   3866 /dev/null
nginx   7989 root    2w   REG              202,1        0  662378 /home/wwwlogs/nginx_error.log
nginx   7989 root    3u  unix 0xffff88003bd83600      0t0  194987 socket
nginx   7989 root    4w   REG              202,1        0  662378 /home/wwwlogs/nginx_error.log
nginx   7989 root    5w   REG              202,1    10349  662379 /home/wwwlogs/access.log
nginx   7989 root    6u  IPv4             194984      0t0     TCP *:http (LISTEN)
nginx   7989 root    7u  unix 0xffff88003bd832c0      0t0  194988 socket
```

图 4-13 lsof 命令查看 Nginx 打开的文件

4.2.5　跟踪数据到达主机所经路由——traceroute

互联网中信息的传送是通过网络中许多段的设备（路由器、交换机、服务器、网关等）从一端到达另一端。每一个连接在 Internet 上的设备一般情况下都会有一个独立的 IP 地址，通过 traceroute 可以知道信息从用户的手机到互联网另一端的主机所经路径。当然数据包每次由某一同样的出发点到达某一同样的目的地走的路径可能会不一样，但基本上来说大部分时候所走的路径是相同的。

traceroute 这个命令在生产环境中用得最多的场景是某个用户反馈无法访问后台，但其他用户访问正常。这时最好让有一定技术基础的用户运行 traceroute 命令来查看访问后台的过程中哪个地方出了错。

使用 traceroute 的结果如图 4-14 所示。

```
[root@jeff ~]# traceroute www.baidu.com
traceroute to www.baidu.com (180.149.131.98), 30 hops max, 60 byte packets
 1  123.57.239.247 (123.57.239.247)  0.815 ms  1.149 ms  6.432 ms
 2  10.87.64.149 (10.87.64.149)  1.130 ms 10.87.64.33 (10.87.64.33)  0.749 ms 10.87.88.37 (10.87.88.37)  7.168 ms
 3  10.87.72.202 (10.87.72.202)  1.192 ms 10.87.64.190 (10.87.64.190)  1.075 ms 10.87.72.202 (10.87.72.202)  1.185 ms
 4  123.56.34.82 (123.56.34.82)  1.067 ms  1.173 ms  1.168 ms
 5  * * *
 6  * * *
 7  180.149.128.66 (180.149.128.66)  5.612 ms 180.149.128.150 (180.149.128.150)  4.858 ms 180.149.128.70 (180.149.128.70)  5.224 ms
 8  180.149.129.6 (180.149.129.6)  2.390 ms  2.511 ms 180.149.129.170 (180.149.129.170)  2.368 ms
```

图 4-14　traceroute 的结果

4.2.6　文件下载/上传工具——“ssh secure shell client”和“lrzsz”

开发人员在服务器中经常需要上传和下载文件，虽然架设一个 FTP 服务器就能满足以上的需求，但是架设 FTP 的步骤烦琐，配置和使用也需要一定的专业知识，使用下面两款工具就能满足上传和下载文件简单的需求。

1. ssh secure shell client

“ssh secure shell client”是一个在 Windows 下通过 ssh 连接服务器的软件，其自带了文件的上传和下载功能。用“ssh secure shell client”连接上 Linux 服务器后，界面如图 4-15 所示。

从 Linux 下载文件，把 Linux 窗口区域内的文件或文件夹拖放到 Windows 窗口区域即可；往 Linux 上传文件，把 Windows 窗口区域内的文件或文件夹拖放到 Linux 窗口区域即可。

同时这个软件也自带了 shell 命令窗口，可通过这个软件对 Linux 进行操作。

2. lrzsz

lrzsz 软件包，在 Linux 里可代替 FTP 上传和下载。安装 lrzsz 只要执行命令：

```
yum -y install lrzsz
```

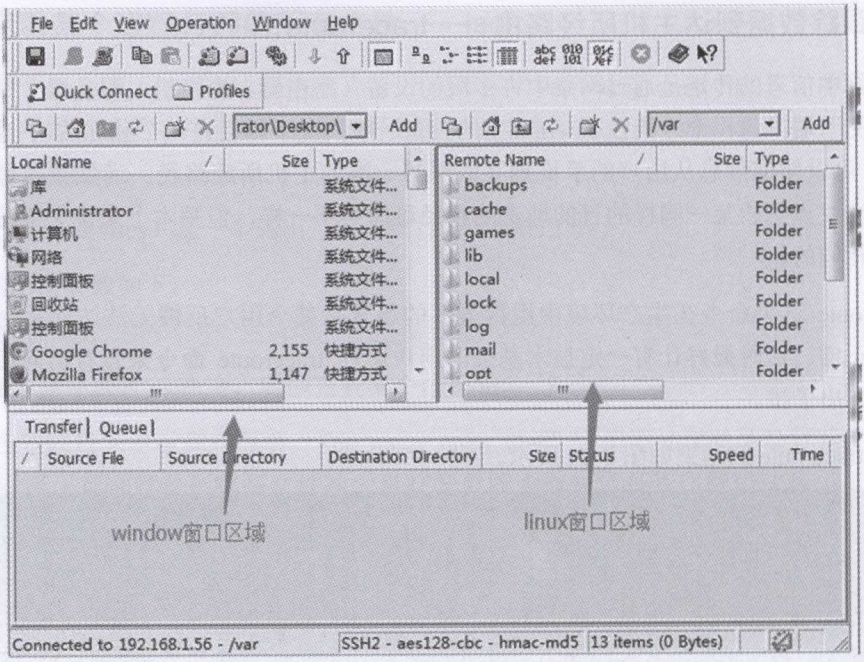


图 4-15 “ssh secure shell client”连接 Linux 服务器后的界面

用户往 Linux 服务器上传文件的时候，在目标文件夹执行命令“rz”，在 ssh 客户端弹出一个窗口让用户选择上传的文件，如图 4-16 所示。

图 4-16 rz 上传文件图

把 Linux 服务器的文件下载到本地，可执行命令“sz 目标文件”，在 ssh 客户端弹出一个文件窗口让用户选择下载文件的保存路径，如图 4-17 所示。

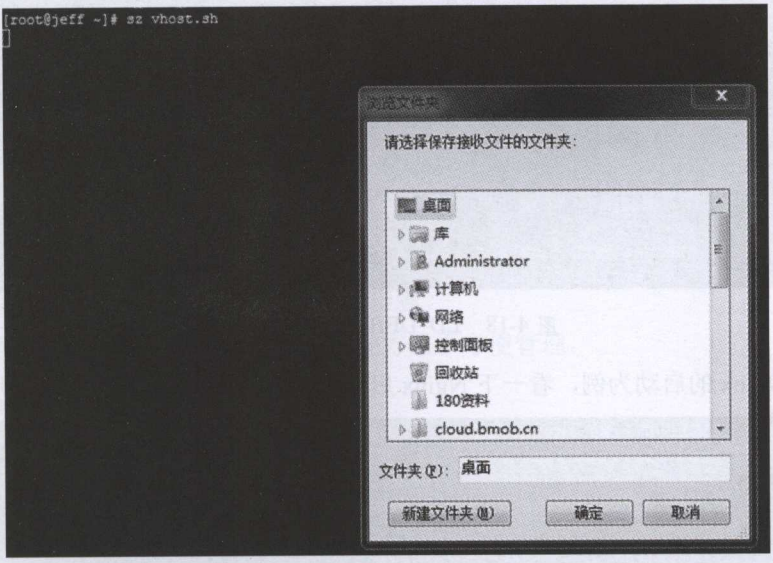


图 4-17　sz 下载文件图

需要注意，某些 ssh 客户端是不支持 sz 和 rz 命令的。

4.2.7　查看程序的依赖库——LD_DEBUG

运行某个程序有时会报以下的错误。

```
error while loading shared libraries: xxxx.so.2:
cannot open shared object file: No such file or directory
```

这是由于查找不到所依赖的库造成的，但从错误信息上看，没法确定所缺的库的具体路径，这就让开发人员无从查起。

对于这种依赖的库找不到的情况，在 Linux 中可以通过设置 LD_DEBUG 环境变量来获得更多的信息。

LD_DEBUG 是 glibc 中的 loader 为了方便自身调试而设置的一个环境变量。通过设置这个环境变量，可以方便地看到 loader 的加载过程。

LD_DEBUG 的详细用法如图 4-18 所示。

```
[root@jeff ~]# LD_DEBUG=help ls
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variable.
```

图 4-18　LD_DEBUG 的详细用法

下面以 Nginx 的启动为例，看一下 Nginx 启动所需要加载的库，如图 4-19 所示。

```
[root@jeff ~]# LD_DEBUG=libs /usr/local/nginx/sbin/nginx -c /usr/local/nginx/conf/nginx.conf
      8923:     find library=libpthread.so.0 [0]; searching
      8923:      search cache=/etc/ld.so.cache
      8923:       trying file=/lib64/libpthread.so.0
      8923:
      8923:     find library=libcrypt.so.1 [0]; searching
      8923:      search cache=/etc/ld.so.cache
      8923:       trying file=/lib64/libcrypt.so.1
      8923:
      8923:     find library=libpcre.so.0 [0]; searching
      8923:      search cache=/etc/ld.so.cache
      8923:       trying file=/usr/local/lib/libpcre.so.0
      8923:
      8923:     find library=libssl.so.10 [0]; searching
      8923:      search cache=/etc/ld.so.cache
      8923:       trying file=/usr/lib64/libssl.so.10
      8923:
      8923:     find library=libcrypto.so.10 [0]; searching
      8923:      search cache=/etc/ld.so.cache
      8923:       trying file=/usr/lib64/libcrypto.so.10
      8923:
      8923:     find library=libdl.so.2 [0]; searching
      8923:      search cache=/etc/ld.so.cache
      8923:       trying file=/lib64/libdl.so.2
```

图 4-19　Nginx 启动所需要加载的库

4.2.8　进程管理利器——superivisor

supervisor 是用 Python 语言编写的基于 Linux 操作系统的一款进程管理工具，用于监控进程的运行，当发现进程闪退时能自动重启。

比如开发人员想在后台运行一个从消息队列中取出消息再发送到邮件的脚本 sendmail.sh，常常会使用&在程序结尾让程序自动运行，并且在退出登录后也继续执行，采用如下的命令。

```
nohupsh /data/sendmail.sh 2>&1 >> /data/logs/sendmail.log &
```

当要把运行 sh 脚本的进程 kill 掉，则需要通过下面两步，如图 4-19 所示。

```
[root@jeff data]# ps aux|grep mail
root      9836  0.0  0.1 106060  1284 pts/0    S    20:00   0:00 /bin/bash /data/sendmail.sh
root      9840  0.0  0.0 103176   868 pts/0    S+   20:00   0:00 grep mail
[root@jeff data]# kill -9 9836
```

图 4-19　kill 掉 sh 脚本的进程

上面管理进程的方法有如下的缺点。

- 不知道进程的状态，不知道进程在运行的过程中是否终止。
- 每次重启进程，kill 进程都需要烦琐的步骤。
- 如果需要运行大量的守护进程，用这种方法不方便管理。

使用 supervisor 管理进程有如下的优点。

- 能自动启动配置好的进程，并监控每个进程的状态，例如进程是运行着还是停止的。
- 监控的进程如果因为各种原因闪退，能自动重启改进程。

安装 supervisor 很简单，使用如下的命令：

```
yum install supervisor
```

supervisor 安装完成后，在“/usr/bin/”目录下增加了两个命令：

- supervisord : supervisor 的服务器端，启动 supervisor 就是运行这个命令。
- supervisorctl：通过该命令和 supervisord 进行交换。

启动 supervisor，先为其创建配置文件/etc/supervisord.conf，内容如下。

```
[unix_http_server]
file=/tmp/supervisor.sock ; (the path to the Socket file)

[supervisord]
logfile=/tmp/supervisord.log ; (main log file;default $CWD/supervisord.log)
logfile_maxbytes=50MB        ; (max main logfile bytes b4 rotation;default 50MB)
logfile_backups=10           ; (num of main logfile rotation backups;default 10)
loglevel=info                ; (log level;default info; others: debug,warn,trace)
pidfile=/tmp/supervisord.pid ; (supervisordpidfile;defaultsupervisord.pid)
nodaemon=false               ; (start in foreground if true;default false)
minfds=1024                  ; (min. avail startup file descriptors;default 1024)
minprocs=200                 ; (min. avail process descriptors;default 200)

[rpcinterface:supervisor]
supervisor.rpcinterface_factory                                              =
```

```
supervisor.rpcinterface:make_main_rpcinterface

[supervisorctl]
serverurl=unix:///tmp/supervisor.sock ; use a unix:// URL  for a unix Socket

[include]
files = /etc/supervisord.conf.d/*.conf
```

在上面的文件中，为了更好地管理进程监控脚本，把进程监控脚本放在/etc/supervisord.conf.d/。例如，为监控运行脚本/data/sendmail.sh，创建文件/etc/supervisord.conf.d/mail.conf，内容如下。

```
   [program:mail]                          ; 在 supervisor 监控列表中的标识
   directory = /data                       ; 启动命令时进入的目录，如果 command 中的
命令没有使用绝对路径，这项一定要设置
   command = /bin/sh /data/sendmail.sh     ; 启动的命令
   autostart = true                        ; 随着 supervisor 启动而启动。
   autoRESTart = true                      ; 自动重启。
   startretries = 10                       ; 启动失败时的最多重试次数
   startsecs = 5                           ; supervisor 启动 5 秒后启动
   user = root                             ; 以 root 用户的身份运行
   redirect_stderr = true                  ; 重定向 stderr 到 stdout
   numprocs=1                              ; 启动 1 个进程
   stdout_logfile = /data/logs/mail.log    ; 输出日志的位置
```

运行 supervisord 服务端程序。

```
   /usr/bin/Python /usr/bin/supervisord -c /etc/supervisord.conf
```

使用 supervisorctl 查看 supervisor 管理进程的状态。

```
   supervisorctl status
   mail                             RUNNING    pid 12663, uptime 0:00:07
```

停止名为“mail”的进程。

```
   [root@ ~]# supervisorctl stop mail
   mail: stopped
```

supervisor 还包含了很多实用的命令，可通过帮助查看这些命令，如图 4-20 所示。

```
[root@ ~]# supervisorctl help

default commands (type help <topic>):
=====================================
add    clear  fg        open  quit    remove  restart   start   stop  update
avail  exit   maintail  pid   reload  reread  shutdown  status  tail  version
```

图 4-20　supervisor 帮助命令

4.3　故障案例分析

1. 进程管理软件引起的最大连接数限制

故障现象：笔者在办公室的电脑上测试服务器上某个程序的最大连接数，发现连接数到达 1000 后全部连接断开。

查找故障：笔者一开始以为是服务程序的 Bug，但在程序上加了 log，发现连接数到达 1000 后内存使用情况正常，程序处理的连接也远远没达到理论上的瓶颈。

既然排除了程序的问题，那只能怀疑是 Linux 的限制。按照网络上的教程，不断优化内核的参数，现象还是如旧。

最后笔者在同机房的另一台服务器上进行这个测试，连接数很顺利地超过了 1000，但在办公室电脑上测试，到了 1000 的连接就断开了。

原因分析：办公室的电脑使用了 supervisor 管理进程，修改了最大连接数的限制后没有重启 supervisor，导致了软件的最大连接数一直受限制。

经验教训：

网络是个整体，任何一个节点上都有可能出现问题。查找问题的时候，要综合考虑。

2. 占满磁盘空间引起网站无法登录的问题

故障现象：在测试环境中，登录网站时用户名和密码都是正确的，但页面跳转后无法保持登录的状态。

查找故障：在笔者电脑的开发环境中登录正常，因此排除是代码引起的这个问题。

从故障现象分析，是登录的状态没有被保存下来。在网站中保存登录的状态，是通过服务器的 Session 和浏览器的 Cookie 实现的。查看服务器保存 Session 的目录，没发现任何异常。

测试人员在测试网站文件上传的功能时，提示文件写入失败。笔者把文件上传失败和 Session 的问题综合考虑：都是写入失败，会不会是硬盘的问题？

结果笔者查看硬盘的空间，发现硬盘空间用完了。把占据大量硬盘空间的日志文件删除后恢复正常。

原因分析：前段时间在测试服务器上开了 DEBUG 日志没有关闭，日志文件过大导致硬盘空间用完，因此网站没法保存新的 Session 而造成无法登录。

经验教训：

养成良好的操作习惯，解决了问题要把 DEBUG 日志关掉。

第 5 章
Nginx——App 后台 HTTP 服务的利器

Nginx 是一个高性能的 HTTP 和反向代理服务器，在 BAT 等巨头和众多的移动互联网公司中有广泛的应用。其主要特点是占用内存少，并发能力强。

本章主要介绍 Nginx 下面的内容。

- 简介
- 基本原理
- 常用配置
- 性能统计
- 实现负载均衡的方案

5.1 简介

Nginx 与 Apache 类似，其是一个高性能的 HTTP 和反向代理服务器，也是一个 imap/pop3/smtp 代理服务器。Nginx 是由 Igor Sysoev 为俄罗斯访问量第二的 Rambler.ru 站点开发的，其已经在该站点运行超过三年。Igor 将 Nginx 源代码以类似 BSD 许可证的形式发布。

到目前为止，Nginx 已经成为一个非常流行的 Web 服务器，在国内外有着众多的用户。各个 Web 服务器的使用情况如图 5-1 所示。

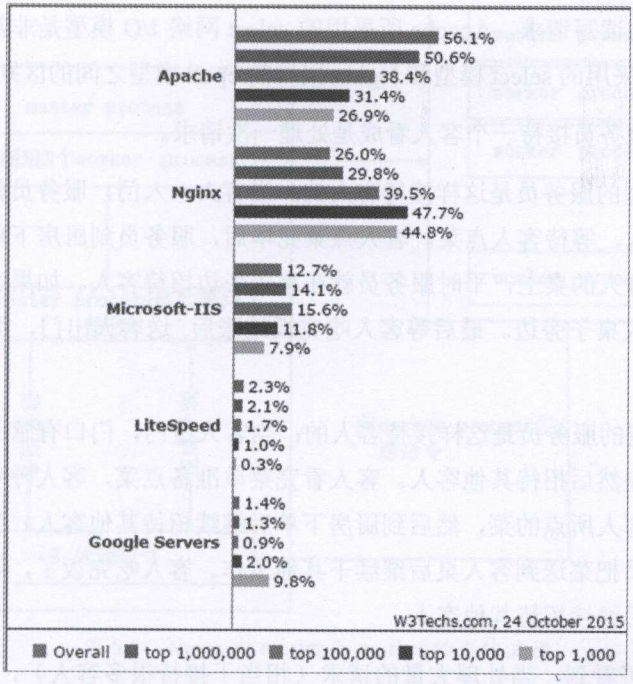


图 5-1　各个 Web 服务器的使用情况（数据出处：http://w3techs.com/technologies/cross/web_server/ranking）

根据图 5-1 的统计结果，全球 Top 1000 的网站中，有 44.8%使用的是 Nginx，全球 Top 10,000 的网站中，有 47.7%使用的是 Nginx。

5.2　基本原理

下面讲述 Nginx 两方面的基本原理。

- 工作模型
- 进程解析

5.2.1　工作模型

Nginx 的高性能主要是其使用了 epoll（使用于 Linux 内核 2.6 版本及以后的系统。在某些发行版本中，如 SuSE 8.2, 有让 2.4 版本的内核支持 epoll 的补丁）和 kqueue（使用于 FreeBSD 4.1+、OpenBSD 2.9+、NetBSD 2.0 和 MacOS X）网络 I/O 模型，而 Apache 则使用的是传统的 select 模型（注：Apache 2.4 版本后也使用了 epoll 网络 I/O 模型）。目前在 Linux 下能够承受高并发访问的著名开源软件 Squid、Memcached 采用的都是是 epoll 网络 I/O 模型。

处理大量连续的读写请求，Apache 所采用的 select 网络 I/O 模型是非常低效的。下面用一个例子分析 Apache 采用的 select 模型和 Nginx 采用的 epoll 模型之间的区别。

在菜馆中，把服务员接待一个客人看成是处理一次请求。

使用 select 模型的服务员是这样接待客人的：当客人一入门，服务员就立刻引导客人入座，然后把菜单递给客人，等待客人点菜。客人点菜完毕后，服务员到厨房下单。当厨房把菜做好后，服务员送菜到客人的桌上，平时服务员就在桌子旁边招待客人。如果客人这时看电影，服务员也要守候在客人桌子旁边。最后等客人吃完饭结账后，送客人出门，服务员继续招待下一个客人。

使用 epoll 模型的服务员是这样接待客人的：当客人进门，门口有感应器通知服务员，服务人就引导客人入座然后招待其他客人。客人看完菜单准备点菜，客人呼唤服务员，服务员来到客人的桌上记下客人所点的菜，然后到厨房下单后继续招待其他客人。当服务员收到厨房通知菜做好了，服务员把菜送到客人桌后继续干其他工作。客人吃完饭了，客人呼唤服务员结账，服务员送客人出门后继续招待其他客人。

从上面的过程可看到，当处理大量的请求（相当于接待很多客人），epoll 模型远远高效于 select 模型。

5.2.2 进程解析

正常工作的 Nginx 有多个进程，基本的有 master process（称为主进程）和 worker process（称为工作进程）。

maste process：充当整个进程组与用户交互的接口，维护并监控 worker process。不处理具体的业务，只把相应的信息发到 worker process。其同时负责 Nginx 的平滑重启、配置文件生效、关闭等。

worker process：处理具体的任务。

Nginx 中 master 和 worker 的工作流程如图 5-2 所示。

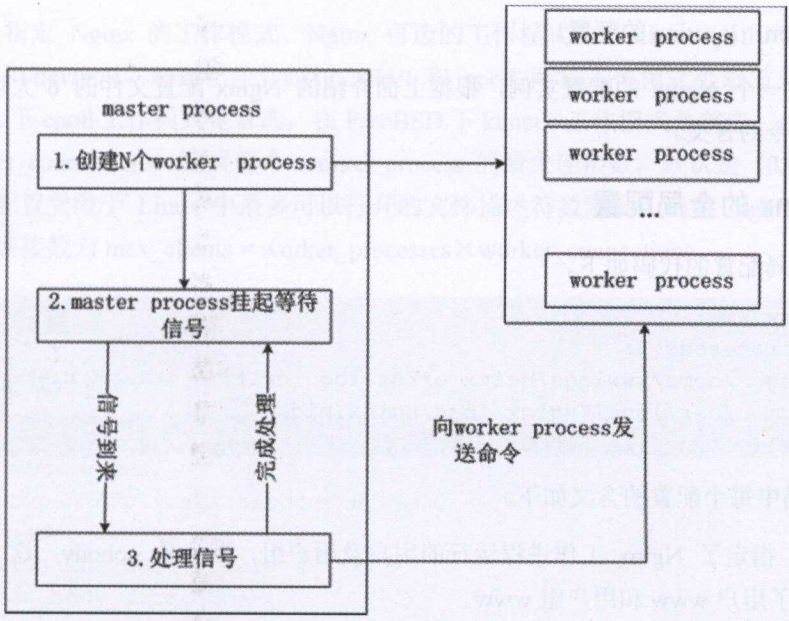


图 5-2　Nginx 中 master 和 worker 的工作流程

5.3　常用配置

Nginx 的配置文件 nginx.conf 是纯文本书件，位于 Nginx 安装目录的 conf 目录下，整个配置文件是以块的形式组织。每个块以“{}”来表示，采用嵌套的方式，一个大块中可以包括小块。最大的块是 main 块，main 块里包含 event 块和 http 块，http 块包含了 upstream 块和 server 块，server 块包含了多个 location 块，整个配置文件的结构如图 5-3 所示。

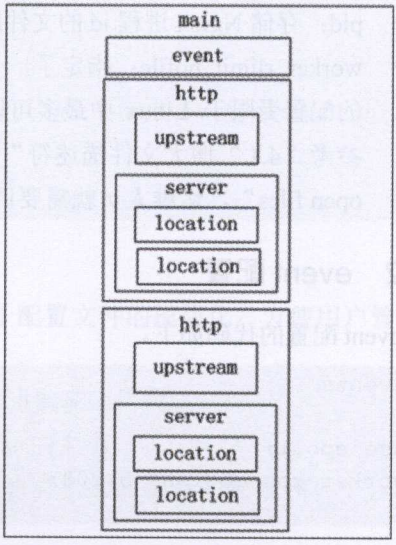


图 5-3　Nginx 配置文件结构图

每个模块的含义如下。

- main：Nginx 的全局属性配置。
- event：Nginx 的工作模式及连接数上限。
- http：http 服务器相关属性的配置。
- upstream：负载均衡属性的配置。
- server：虚拟主机的配置。

- location：location 的配置。

下面通过一个 Nginx 的配置实例，根据上面介绍的 Nginx 配置文件的 6 大块，详细介绍 Nginx 每个指令的含义。

5.3.1 Nginx 的全局配置

Nginx 全局配置的代码如下。

```
user  www www;
worker_processes 4;
error_log  /home/wwwlogs/nginx_error.log  crit;
pid        /usr/local/nginx/logs/nginx.pid;
worker_rlimit_nofile 52000;
```

这段代码中每个配置的含义如下。

- user：指定了 Nginx 工作进程运行的用户及用户组，默认是 nobody，这个配置文件是使用了用户 www 和用户组 www。
- worker_processes：指定 Nginx 开启的工作进程数。每个进程大约占用 10~12MB 的内存。如果是多核的 CPU，这里应设置和 CPU 核数一样的进程数。
- error_log：全局错误日志的位置与日志输出的级别。日志的输出级别可选择 debug、info、notice、warn、error、crit，其中 debug 级别输出的日志最详细。当运维人员查找问题时，错误日志是非常重要的参考。
- pid：存储 Nginx 进程 id 的文件路径。
- worker_rlimit_nofile：指定了一个 Nginx 进程最多可以打开的文件描述符。注意，这里的配置受限于 Linux 中最多可以打开的文件描述符配置。关于 Linux 配置的详解，请参考“4.1.2 增大文件描述符”。如果 Nginx 的错误日志中出现错误提示“Too many open files”，运维人员就需要调整这个值了。

5.3.2 event 配置

event 配置的代码如下。

```
events
{
use epoll;
worker_connections 51200;
}
```

这段代码中每个配置的含义如下。

- use：指定 Nginx 的工作模式。Nginx 可选的工作模式有：select、poll、kqueue、epoll、rtsig 和/dev/poll。前面已经介绍过 select 和 epoll 两种工作模式处理方式的不同。在 Linux 下 epoll 工作模式是首选，在 FreeBSD 下 kqueue 工作模式是首选。
- worker_connections：定义每个 worker process 的最大连接数，默认是 1024。注意，这里的配置受限于 Linux 中最多可以打开的文件描述符数限制。当前 Nginx 可以处理的最大连接数为 max_clients = worker_processes × worker_connections。

5.3.3　http 配置

http 配置的代码如下：

```
http{
include mime.types;
default_type  Application/octet-stream;
client_header_buffer_size 32k;
large_client_header_buffers 4 32k;
client_max_body_size 50m;
sendfile on;
tcp_nopush     on;
keepalive_timeout 60;
tcp_nodelay on;

gzip on;
gzip_min_length  1k;
gzip_buffers     4 16k;
gzip_http_version 1.0;
gzip_comp_level 2;
gzip_types   text/plain Application/x-javascript text/css Application/xml;
gzip_vary on;
include vhost/*.conf;
}
```

这段代码中每个配置的含义如下。

- include：包含其他的配置文件，这种机制有利于配置文件的模块化，方便用户管理大量的配置文件。
- default_type：当文件类型未定义时，默认使用二进制流的格式。
- client_header_buffer_size：客户端请求头 buffersize 的大小。
- large_client_header_buffers：客户端请求中较大的消息头的缓存的数量和大小，这里“4”是数量，“32k”是大小。
- client_max_body_size：客户端请求中 http body 的大小，一般可以理解为请求的文件大小。

- sendfile：设置为 on 表示启动高效传输文件的模式。sendfile 可以让 Nginx 在传输文件时直接在磁盘和 tcp Socket 之间传输数据。如果这个参数不开启，会先在用户空间申请一个 buffer，用 read 函数把数据从磁盘读到 cache，再从 cache 读取到用户空间的 buffer，再用 write 函数把数据从户空间的 buffer 写入到内核的 buffer，最后到 TCP Socket。开启这个参数后，可以让数据不用经过用户 buffer。

设置 sendfile 为 off 时传输文件的流程，如图 5-4 所示。

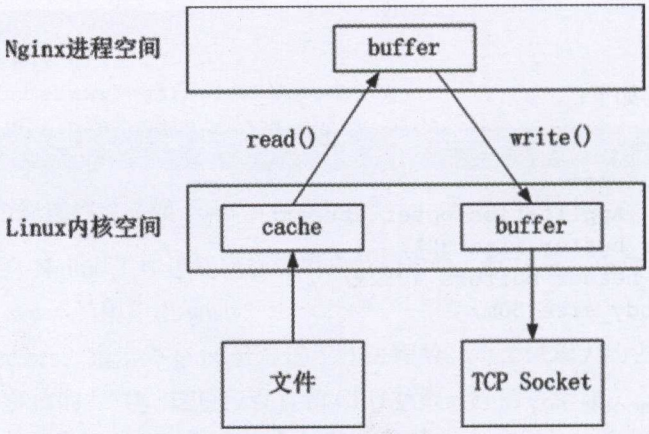


图 5-4　设置为 sendfile off 时传输文件的流程

设置 sendfile 为 on 时传输文件的流程，如图 5-5 所示。

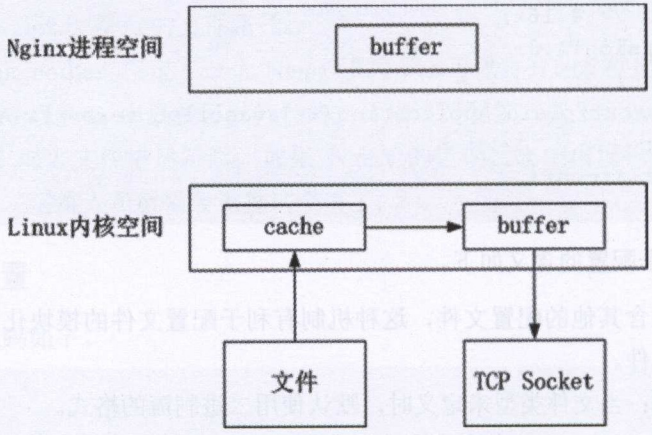


图 5-5　设置为 sendfile on 时传输文件的流程

- tcp_nopush：该选项仅在 sendfile 开启的时候才起作用，主要防止网络堵塞。
- keepalive_timeout：设置客户端保持活动连接的时间。超过这个时间，服务器会关闭连接。

- 下面是 Nginx 的 httpgzip 模块的配置，这个模块支持在线实时压缩输出数据流，需要在编译 Nginx 时带上参数“--with-http_gzip_static_module”才能使用这个模块。
- gzip：设置为 on，启动 gzip 模块。
- gzip_min_length：设置只有当页面的大小大于这个值时，才启用 gzip 压缩。页面大小值通过读取 http 头“Content-Length”来获取。建议是 1KB，文件太小，压缩后有可能会更大。
- gzip_buffers：gzip 的缓冲区的数量和大小。默认是申请和“Content-Length”中一样大小的缓冲区。
- gzip_http_version：支持的 HTTP 协议版本。
- gzip_comp_level：用 gzip 压缩比。取值是从 1~9，1 是压缩比最低，但速度快，9 是压缩比最高，但速度慢，而且特别消耗 CPU 资源。
- gzip_types：所压缩文件的类型。一般来说是压缩传输中的文本资源文件，例如 CSS、JS、HTML 等。
- gzip_vary：是否让前端的缓存服务器缓存压缩后的 GZIP 文件。
- include vhost/*.conf：包含 vhost 文件夹中后缀名为“conf”的配置文件。通常在 vhost 文件夹下存放的是内容为 server 块的 conf 文件，根据经验，最好是一个域名对应一个 conf 文件，以方便管理。

5.3.4　负载均衡配置

负载均衡配置的代码如下。

```
upstream test.com{
server 192.168.1.20:80 weight=2;;
server 192.168.1.21:80 weight=1;
}
```

upstream 模块通过简单的调度算法实现客户端到服务器的负载均衡。在上面的例子中，test.com 是这个负载均衡的名字，可以在后面的配置中调用。

Nginx 支持以下 4 种负载均衡算法。

- 加权轮询（默认的算法）：请求按时间分别分配到不同的服务器上。
- ip_hash：使用请求的 ip 算出 hash 值，根据 hash 值分配到不同的服务器上，固定的 ip 的请求，会分配到固定的服务器。这种策略有效地解决了网站服务的 session 共享问题。
- fair：按后端服务器的响应时间来分配请求，响应时间短的优先分配。Nginx 默认是不支持这种负载均衡算法，需要安装 Nginx 模块和 upstream_fair 模块。
- url_hash：使用请求的 URL 算出 hash 值，根据 hash 值分配到不同的服务器上，固定的

URL 的请求，会分配到固定的服务器上。这种策略有利于提高后端服务器的缓存命中率。Nginx 默认是不支持这种负载均衡算法，需要安装 Nginx 的 hash 软件包。

upstream 模块可以为所配置的服务器指定状态值，常用的状态值如下。

- down：服务器不参与到负载均衡中，当后台人员进行故障排查时这个状态非常有用。
- weight：制定轮询的权重，权重越大，分配到的几率越多。在上面的例子中，根据权重的不一样，分配到 20 和 21 的请求的比例大概是 2:1。
- backup：备份机器。当其他的服务器不可用时，才把请求分配到这台服务器。
- max_fails：允许请求失败的次数，默认值是 1。
- fail_timeout：经历了 max_fails 次失败后，暂停服务的时间。

注意：当负载均衡是 ip_hash 时，服务器的状态值不能是 backup 和 weight。

5.3.5 server 虚拟主机配置

server 虚拟主机配置的代码如下。

```
server
{
listen       80;
server_name local.test.cn;
index index.html index.htm index.php default.html default.htm default.php;
root  /var/www/test;
```

这段代码中每个配置的含义如下。

- listen：指定虚拟主机监听的端口。
- server_name：指定虚拟主机对应的域名，多个域名之间以空格分割。
- index：默认的首页文件。
- root：网站的根目录。

5.3.6 location 配置

location 配置的代码如下。

```
location ~ .*\.(gif|jpg|jpeg|png)$
{
  expires      30d;
}
```

location 支持正则表达式和条件判断匹配，用户可以通过 location 指令对动、静态网页进行过滤处理。

上面这段代码的含义是经过正则表达式匹配，设置文件格式为 GIF、JPEG、PNG 的文件在 HTTP 应答中“Expires”和“Cache-Control”的 HTTP 头，以达到在浏览器中缓存图片的作用。这里表示把图片在浏览器中缓存 30 天。

5.3.7　HTTPS 的配置

App 经常需要通过 HTTPS 协议来访问某些对安全性很高要求的 API（例如登录、注册）。HTTPS 核心的是安全证书，生成安全证书有两种途径。

- 缴纳一定的费用，到证书服务商申请。
- 用户给自己颁发证书，即手动生成。

如果证书只是用在开发阶段，那么给自己颁发证书就行了，没必要到证书服务商那申请，不划算。下面介绍怎么手动生成证书，并在 Nginx 中配置使用这个生成的证书。

在 CentOS 环境下，生成证书前先要确保安装 openSSL 和 openSSL-devel，如果没安装，使用下面的命令安装。

```
yum install openssl
yum install openssl-devel
```

生成证书的代码如下。

```
cd /usr/local/nginx/conf
opensslgenrsa -des3 -out local.key 1024
openssl req -new -key local.key -out local.csr
opensslrsa -in local.key -out local_nopwd.key
openssl x509 -req -days 365 -in local.csr -signkey local_nopwd.key -out
local.crt
```

在 Nginx 的虚拟主机中加上下面的配置，并把端口设置为 443，就能使用 https://api.test.cnm 的形式访问需要通过 HTTPS 加密的 API，配置如下。

```
server {
    listen 443;
    ssl on;
    ssl_certificate  /usr/local/nginx/conf/local.crt;
    ssl_certificate_key  /usr/local/nginx/conf/local_nopwd.key;
    server_name api.test.cn;
    index index.html index.htm index.php default.html default.htm default.php;
    root  /var/www/test;
}
```

5.3.8 下载 App 的配置

APK 和 IPA 分别是 Android 应用和 iOS 应用的扩展名。如果浏览器下载 Nginx 服务器中后缀名为.apk 和.ipa 的文件时，浏览器会自动重命名为 ZIP 文件。

如果需要下载时文件名后缀就是.apk 或.ipa，可以修改 Nginx conf 目录下的 mime.types 文件，在文件中增加下面的两行。

```
application/vnd.android.package-archive apk;
application/iphone pxl ipa;
```

重启 Nginx 后配置生效。

5.3.9 生产环境中修改配置的良好习惯

修改配置文件前，务必要养成先备份配置文件的习惯。

如果修改配置文件后需要重启 Nginx，在重启 Nginx 前，先使用- t 参数检查 Nginx 的配置文件语法是否正确，如图 5-6 所示。

```
[root@jeff ~]# /usr/local/nginx/sbin/nginx -t
nginx: the configuration file /usr/local/nginx/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/nginx/conf/nginx.conf test is successful
```

图 5-6 Nginx- t 参数检查 Nginx 的配置文件

如果重启 Nginx 前没保证配置文件语法正确，当 Nginx 重启过程中检测到配置文件有问题，Nginx 就会停止服务，这样就影响网站的正常运行。

当屏幕显示配置文件语法正确后，用下面的命令平滑重启 Nginx。

```
[root@mode ~]# /usr/local/nginx/sbin/nginx -s reload
```

上面的命令可以让 Nginx 先完成正在处理的请求后再重启 Nginx 的服务，使用户体验更好。

5.4 性能统计

编译 Nginx 源码的时候带上参数“--with-http_stub_status_module”，就安装了 Nginx 的统计模块，这个模块能够获取 Nginx 自上次启动以来的工作状态。

在虚拟主机的配置文件中添加以下的代码，启动统计功能。

```
location /nginx_status {
stub_status on;
```

```
    access_log   off;
    }
```

重启 Nginx 使配置文件生效，就能在浏览器中输入“http://域名/nginx_status”了解 Nginx 的状态，会出现类似下方的信息。

```
Active connections: 1
server accepts handled requests
653 653 685
Reading: 1 Writing: 1 Waiting: 0
```

上面每项的含义如下。

- Active connections：当前 Nginx 正处理的活动连接数。
- server accepts handled requests：共处理了 653 次连接，共处理了 653 次握手，共处理了 685 次请求。
- Reading：Nginx 读取到客户端的 Header 信息数。
- Writing：Nginx 返回给客户端的 Header 信息数。
- Waiting：开启 keep-alive 的情况下，这个值等于 Active - (Reading + Writing)，是 Nginx 已经处理完成，正在等候下一次请求指令的驻留连接。

所以当 App 的请求被快速处理完毕的情况下，Waiting 数比较多是正常的。如果 reading+writing 数目较多，则说明 App 后台并发访问量大，Nginx 正在处理过程中。

5.5　实现负载均衡的方案

应用服务器上处理业务逻辑，由于应用服务器地位上的重要性，为了保证应用服务器高可用，可使用 Nginx 负载均衡和健康检查特性，一个初级的方案如图 5-7 所示。

在上面的方案中，负载均衡服务器部署了 Nginx，其绑定了两个 ip，外网的 ip 和内网 ip。用 DNS 服务把域名绑定到外网 ip，通过内网 ip 和应用服务集群内的服务器通信。

应用集群内的机器不能直接访问网络，所有数据的进出都要经过负载均衡服务器。

这个方案有以下两个好处。

- 保证了应用服务的高可用，即使有台应用服务器宕机，其他服务器也会继续工作。
- 应用服务器不直接连接 Internet，减少了被入侵的可能性。

负载均衡服务器上面部署了 Nginx，使用其负载均衡和健康检查特性。

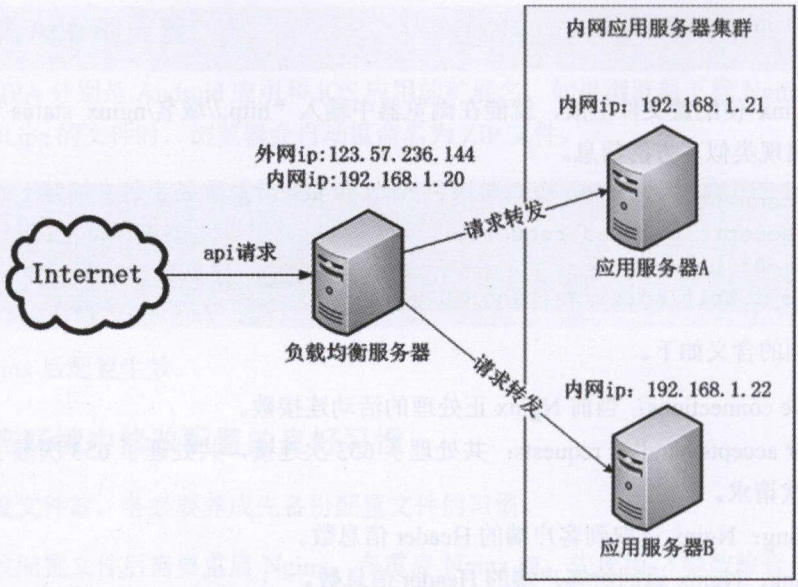


图 5-7　Nginx 负载均衡方案

这个方案还有个问题：负载均衡服务器只有一台，如果负载均衡服务器宕机，整个服务就不可用。

现在业界普遍解决 Nginx 高可用的方法是 Nginx+Keepalived，部署两台 Nginx 服务器，通过 Keepalived 把外网 ip 绑定到一台 Nginx 服务器上，如果这台 Nginx 服务器宕机，Keepalived 就把这个 ip 漂移到另外一台 Nginx 服务器上，使服务不受影响。

创业团队中的运维大多数是由开发人员兼职，非专业运维人员实现 Nginx+Keepalived 方案，需要挺高的学习成本和冒一定的风险，而且这种方案中有一台服务器平时是处于闲置状态，资源利用率不高。笔者一向倡导创业团队的架构原则是**“尽量使用成熟可靠的云服务和开源软件，自身只专注于业务逻辑”**。现在云服务器提供了负载均衡 SLB 的服务。使用云服务器上负载均衡 SLB 的服务有以下两个好处。

- 云服务器上有专业的运维团队保证负载均衡 SLB 的高可用。
- 负载均衡 SLB 的服务，比自己购买服务器搭建负载均衡服务便宜多了（甚至有免费的服务，现在 UCloud 的负载均衡就是免费的）。

花钱购买可靠成熟的服务，不但节省资源，还能提高开发的效率，把自身的精力专注于最核心的业务上。

5.6　用 Nginx 处理业务逻辑

在一般的 App 后台架构中，Nginx 是不处理任何业务逻辑的。例如经典的 LNMP 架构中，客户端请求到达 Nginx 后，Nginx 通过查找 location 命令，将所有以“.php”为后缀的文件都交给 PHP 处理。

为了弥补 Nginx 不能处理业务的缺陷，Nginx 开源社区的开发者给 Nginx 添加了 Lua 模块，Nginx 使用 Lua 模块后就具备了处理业务的能力。

Lua 是一种脚本语言，由巴西里约热内卢天主教大学里的一个研究小组于 1993 年开发。其设计目的是为了嵌入到应用程序中，从而为应用程序提供更灵活的扩容功能。Lua 由标准 C 编写而成，几乎在所有操作系统和平台上都可以编译、运行。某些应用和游戏使用了 Lua 作为嵌入式脚本语言以增强自身的扩展性，例如《魔兽世界》、《愤怒的小鸟》。同时 Lua 也是一门轻量级的脚本语言，其官方版本只包括一个精简的核心和最基本的库。

在业界广泛使用的 OpenResty 项目把 Lua 语言嵌入 Nginx 中，同时其集成了大量实用的模块以方便开发人员使用。使用 OpenResty 的开发人员可以用 Lua 对 Nginx 进行脚本编程，从而可以在 Nginx 请求处理阶段执行各种 Lua 代码来处理业务逻辑。OpenResty 的开发者章亦春（网名 agentzh），其同时也在维护很多 Nginx 模块，如果读者需要学习开发 Nginx 模块，他博客上的教程不得不看，地址为：https://github.com/openresty/Nginx-tutorials/tree/master/zh-cn。

下面是 Nginx 添加了 Lua 模块后一个例子。

```
location /lua {
    set $test "hello, world.";
    content_by_lua '
        ngx.header.content_type = "text/plain";
        ngx.say(ngx.var.test);
    ';
}
```

把上面的代码加入到 localhost 的配置文件中，在浏览器输入网址 http://localhost/lua，就能看到浏览器中输出字符串“hello, world.”。

Nginx+Lua 模块一般是处理一些逻辑比较简单的业务，复杂的业务处理还是在应用服务器上。

Nginx+Lua 除了能处理简单的业务外，还能有以下的用途。

- 统计所有慢请求。
- 过滤不合法的请求。

第 6 章
MySQL——App 后台最常用的数据库

数据存储是 App 后台中必不可少的功能，MySQL 作为一个久经考验的数据库软件，数据存储功能卓越。因此熟练运用 MySQL 是每个 App 后台开发人员必须掌握的技能。

本章主要介绍 MySQL 下面的内容。

- 基本架构
- 配置文件详解
- 优化（软件，硬件，架构）
- SQL 慢查询分析
- 云数据库简介

6.1 基本架构

MySQL 在架构上分为三层。

- 服务层：大多数基于网络的客户端/服务端工具都有这一层，这一层主要是处理连接和安全验证。
- 核心层：这层处理 MySQL 的核心业务。
  - 查询分析，优化，缓存和内置的函数。
  - 内建的视图，存储过程，触发器。

- 存储引擎层：存储引擎负责数据的存储和提取。核心层通过存储引擎的 API 与存储引擎通信，这样子就遮蔽了不同存储引擎的差异，使得这些差异对上层查询是透明的。存储引擎之间不会相互通信，只是简单地响应上层的查询。

MySQL 的整体架构如图 6-1 所示。

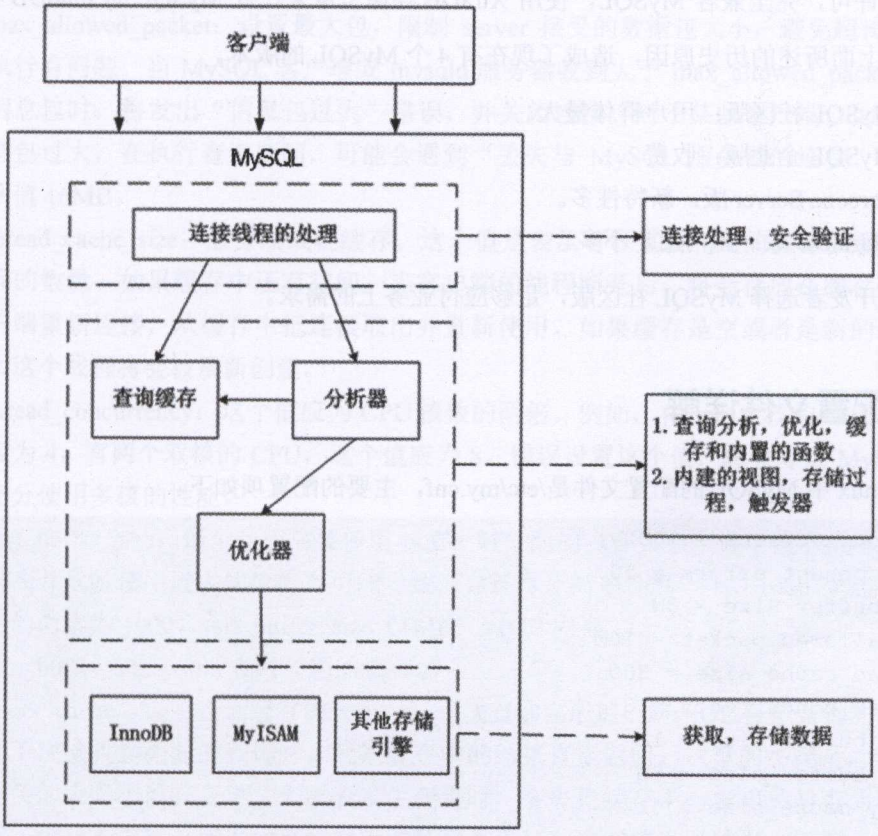


图 6-1　MySQL 的整体架构

6.2　选择版本

MySQL 的发展经历了下面的几个里程碑。

- 1979 年，创始人 Monty Widenius 写了最初的版本。
- 1996 年，发布 1.0 版本。
- 1995-2000 年，MySQL AB 公司成立，引入 BDB。

- 2000 年，集成 MyISAM 和 Replication。
- 2008 年，MySQL AB 被 Sun 收购，2009 年推出 5.1 版本。
- 2009 年，Oracle 收购 Sun，2010 年 12 月推出 5.5 版本。

MySQL 被 Oracle 收购后，MySQL 创始人 Monty Widenius 主导开发了 MariaDB，采用 GPL 授权许可，完全兼容 MySQL，使用 XtraDB 存储引擎来代替 MySQL 的 InnoDB 存储引擎。

由于上面所述的历史原因，造成了现在有 4 个 MySQL 的版本。

- MySQL 社区版：用户群体最大。
- MySQL 企业版：收费。
- Percona Server 版：新特性多。
- MariaDB 版：国内用户不多。

建议开发者选择 MySQL 社区版，足够应付业务上的需求。

6.3 配置文件详解

在 Linux 中 MySQL 的配置文件是/etc/my.cnf，主要的配置项如下。

```
max_connections = 1000
max_connect_errors = 50
key_buffer_size = 3M
max_allowed_packet = 16M
thread_cache_size = 300
thread_concurrency = 8
sort_buffer_size = 1M
join_buffer_size = 8M
query_cache_size = 512M
query_cache_limit = 2M
read_buffer_size = 2M
read_rnd_buffer_size = 16M
myisam_sort_buffer_size = 8M
innodb_buffer_pool_size = 4G
innodb_log_file_size = 128M
innodb_log_buffer_size = 8M
innodb_flush_log_at_trx_commit = 1
innodb_lock_wait_timeout = 50
```

- max_connections：MySQL 所允许的同时会话数的上限。即使到了最大连接数的上限，其中一个连接将会被保留作为管理员登录所用。如果该数值设置过少，很容易出现错

误“Too many connections”。

- max_connect_errors：每个客户端连接的最大错误允许数，如果达到了这个数的限制，这个客户端将会被 MySQL 服务阻止直到执行“FLUSH HOSTS”或者服务重启。
- key_buffer_size：关键词缓冲区大小，用来缓冲 MyISAM 表的索引块，它决定了数据库索引处理的速度，尤其是读取索引的速度。
- max_allowed_packet：设置最大包，限制 server 接受的数据包大小，避免超长 SQL 的执行有问题。当 MySQL 客户端或 mysqld 服务器收到大于 max_allowed_packet 字节的信息包时，将发出“信息包过大”错误，并关闭连接。对于某些客户端，如果通信信息包过大，在执行查询期间，可能会遇到“丢失与 MySQL 服务器的连接”错误。默认值 16MB。
- thread_cache_size：服务端线程缓存。这个值是表示可以重新利用的保存在缓存中的线程的数量。如果缓存中还有空间，当客户端的线程断开后，将会存放在缓存中，当客户端重新连接，从缓存中把连接取出并重新使用。如果缓存是空或者是新的请求，那么这个线程将会被重新创建。
- thread_concurrency：这个值应为 CPU 核数的两倍。例如，有一个双核的 CPU，这个值应为 4，有两个双核的 CPU，这个值应为 8。错误设置这个值，将会导致 MySQL 不能充分使用多核的性能。
- sort_buffer_size：每个连接需要使用 buffer 时分配的内存大小。这个值不是越大越好，当高并发时使用过大的值，有可能导致内存耗尽。简单计算一下，1000 个连接时所使用的内容为 1000×sort_buffer_size（1MB）=1GB 内存。
- join_buffer_size：join 表时使用的缓存。
- query_cache_size：查询缓存的大小。当一次查询完成后，MySQL 会把查询的结果缓存，当下次接收到相同的查询时，就把缓存中的结果直接返回。这里的关键是，缓存期间相关的表必须没有变更。如果表发生变更时，会先把缓存中与表相关的查询设置为无效，再写入更新。所以如果数据库的场景是多写，这个值设置过大，反而会影响写入的效率。
- query_cache_limit：单次查询缓冲区的大小。
- read_buffer_size：设置做 MyISAM 表全表扫描时的缓冲大小。如果在数据库中全表扫描的操作非常频繁，而且没法通过添加索引来优化，那么可以增大这个值来优化性能。
- read_rnd_buffer_size：在排序后从排序好的数据读取行时，行数据将会从这个缓冲区读出。合理设置这个值，将会提升 order by 的性能。注意：MySQL 将会为每个客户端连接申请该缓冲区，并发过大时，该值过大会造成内存开销过大。
- myisam_sort_buffer_size：MyISAM 表发生变化时重新排序所需的缓存。

- innodb_buffer_pool_size：InnoDB 使用缓存来保存索引和原始数据，这个值就是设置缓存的大小。正确地设置这个值，可以有效减少读取数据所需要的磁盘 I/O。
- innodb_log_file_size：数据日志文件的大小。设置更大的值可以提高性能，但也增加了恢复故障数据库的时间。
- innodb_log_buffer_size：日志文件的缓存。增大该值可以提高性能，但也增大了忽然宕机时损失数据的风险。
- innodb_flush_log_at_trx_commit：当执行事务时，会往 InnoDB 存储引擎的日志缓存里面插入事务日志；当事务提交时，必须将存储引擎的日志缓冲写入磁盘，也就是写数据前，需要先写日志。这种方式称为“预写日志方式”。设置为 0 表示每秒日志缓存写入日志文件，日志文件实时写入磁盘；设置为 1 表示日志缓存实时写入日志文件，日志文件实时写入磁盘；设置为 2 表示日志缓存实时写入日志文件，每秒日志文件写入磁盘。
- innodb_lock_wait_timeout：在被回滚前，一个 InnoDB 的事务应该等待一个锁被批准多久。InnoDB 自动检测其事务是否死锁并自动回滚，但是使用了 LOCK TABLES 指令，或者在事务中使用了 InnoDB 外的存储引擎，那么这个死锁可能发生而 InnoDB 无法检测到。这时这个 timeout 值对解决这个问题非常有用。

6.4 软件优化

在下面的章节中，介绍了通用的 MySQL 软件层面优化的方法。

6.4.1 正确使用 MyISAM 和 InnoDB 存储引擎

MyISAM 和 InnoDB 是 MySQL 最常用的两个存储引擎。在 MySQL 5.1 版本中，默认的存储引擎是 MyISAM。到了 MySQL 5.5、5.6 版本，默认的存储引擎是 InnoDB。只有了解了 MyISAM 和 InnoDB 各自的特点，才能合理地使用这两种存储引擎。

MyISAM 基于 ISAM（索引顺序访问方法），支持全文索引，但并非是事务安全，不支持外键，使用表级锁。每个 MyISAM 表存有 3 个文件：FRM 文件存放表结构，MYD 文件存放数据，MYI 存放索引。

InnoDB 是事务型存储引擎，其支持行锁，InnoDB 表的行锁也不是绝对的，如果它在执行一个更新的语句时没法确定更新的范围，也会锁表，例如执行语句 update table set age=3 where name like "%jeff%"，在这个更新语句就会锁表。InnoDB 支持回滚、崩溃恢复、ACID 事务控制，InnoDB 存储它的表和索引在一张表空间，表空间可以包含多个文件。

MyISAM 和 InnoDB 的主要区别如下。

- MyISAM 支持表锁，InnoDB 支持行锁。
- MyISAM 是非事务安全型，InnoDB 是事务安全型。
- MyISAM 不支持外键，InnoDB 支持外键。

6.4.2　正确使用索引

什么是索引？

读者阅读这本书的时候，目录中列出了这本书的所有章节的页码，如果读者在目录中看到感兴趣的章节，根据页码可以很快地翻到对应的章节，无须从头到尾一页一页地翻看。

书的目录就是索引的一种，其能帮助读者快速地查找到目标内容。

同样的，数据库中的索引，也能快速地查找到相应的数据。

下面是使用索引的一些原则。

- 给合适的列建立索引。在 where 子句中经常需要给检索的列建立索引，或者给连接子句中指定的列建立索引，而不是给 select 选择列表中的列建立索引。
- 索引列的值尽可能不同。对于有唯一性的值，索引的效果最好；如果有大量的重复值，索引的效率很差。
- 使用短索引。对字符类型的列建立索引，只要有可能，都应该指定前缀长度。例如，有一个 char（50）的列，如果前 20 或 30 个字符内，多数值是唯一的，那么就不要对整个列进行索引。较小的索引，索引缓存中能容纳更多，消耗的磁盘 IO 更小，能提高查找的速度。
- 利用最左前缀。创建一个 n 列索引，本质上是 MySQL 创建了 n 个索引。多个索引可以起 n 个索引的作用，可以用索引中最左列的值来匹配，这样的列值叫做最左前缀。
- 使用 like 查询时索引会失效。因此尽量少使用 like 查询，对于百万、千万级的数据，如果真的要使用 like 查询，请用专业的搜索软件来实现，例如第 3 章中介绍的 Sphinx，就能很方便地结合 MySQL，快速实现这种海量级数据的查询。
- 不能滥用索引。索引并不是越多越好，使用索引需要恰到好处。过度使用索引会有下面的问题。
  - 索引会占用额外的磁盘空间，降低性能。
  - 当更新数据时索引必须更新。因此索引越多，需要花费在更新的时间上更长。如果在某个长期不用的字段上建立索引，会明显降低更新的速度。
  - 在本章的 MySQL 架构图中可看到，SQL 在执行一个查询语句前会对这个查询语句

进行优化，确定使用哪些索引。滥用索引有可能使 MySQL 选择到不是最优的索引，同时增加了查询优化的时间。

6.4.3 避免使用 select *

当需要返回某个查询结果的列时，为了编写代码的方便，有很多程序员使用“select *”。读者想想，如果结果集有 30 列，而应用只需要其中的 20 列，在 select 语句中写 20 列的列名，还要确保每个列名正确无误，这是多么费劲的事情。

使用“select *”有下面的坏处：

- 在 select 语句执行的过程中，“select *”从数据库中返回的结果更多，降低了查询的速度。
- 过多的返回结果会增大服务器返回给 App 端的数据的传输量。在移动互联网传输速度慢（2G、3G 的情况）、弱网络环境（在不同的建筑中切换，在高速移动中经常断网）下，过大的传输量很容易造成请求的失效。

笔者记得做第一个 App 的时候，有个 API 是返回类似于新浪微博中的个人主页的内容，当时为了方便，就使用了“select *”，一开始返回的字段还比较少，没发现问题，随着功能不断增多，返回的字段越来越多。测试人员反馈这个 API 的访问速度很慢，笔者一看这个 API 的返回数据居然达到了 10 多 KB，这 10 多 KB 的纯文本大小还不包括图片，返回这么多数据响应不慢才怪！最后在查询语句的 select 中只返回了需要的字段，这个 API 的访问速度就恢复正常了。

6.4.4 字段尽可能地设置为 NOT NULL

请读者思考：对于字符串类型来说，“”值和 NULL 值在 App 的显示上有什么区别呢？如果产品上规定必须严格区分一个字符的未填和为空两种状态，那么 NULL 是必需的。如果产品上没有这个规定，那么建议数据库上所有字段都设置为 NOT NULL，必须有默认值。

举个例子：用户名的字段是{“realname”：“xxx”}，如果用户名为空，则应该返回{“realname”：“”}。如果返回值是一个 array，空数据则返回一个空 array。

读者不要以为 null 值不需要空间，其实 null 值是需要占用额外的空间的，并且在查询比较的时候，程序会得更复杂。

null 值给程序带来了额外的开发成本。例如，字符串的字段允许 null 值，那就意味着程序员要处理字符串类型为空和 null 两种情况，App 客户端程序员的工作量就大了，其必须特殊处

理字段的 null 值。如果再加上有的字段有 null 值，有的字段没 null 值，这些情况都需要处理。App 客户端的开发语言 Java 和 Objective-C 都是强类型语言，null 值和空值是不一样的类型，不处理 null 值很容易造成 App 的闪退。在现代互联网高速迭代的背景下，一个 App 开发周期就只有几个月，用简单的方法保证 App 的稳定性显得特别重要。

6.5　硬件优化

使用了软件优化的方法后 MySQL 的性能还没有明显的提升，读者这时就要考虑 MySQL 硬件层面的优化。

6.5.1　增加物理内存

MySQL 读写数据最大的性能瓶颈就是磁盘 IO，从减少磁盘 IO 方面提高性能是个重要的方向。通过加大物理内存可以采取提升文件系统性能，减少磁盘 IO。

Linux 内核在内存中开辟了缓存（系统缓存和应用缓存）来存放数据。

- 当写文件的时候，通过文件延迟写入机制，先把文件保存在缓存，当达到一定的条件（缓存达到多少的百分比或者接到了 sync），才真正写入硬盘。
- 当读文件的时候，会把读出的文件放入缓存，当下次需要读取同样的文件时，就先从缓存中取，如果缓存没有，再从硬盘中读取。

另外 MySQL 中也用了大量的内存来提高性能，例如在“6.3 配置文件详解”中提到的下面这些配置参数。

- thread_cache_size：服务端线程缓存。
- sort_buffer_size：每个连接需要使用 buffer 时，分配的内存大小。
- query_cache_size：查询缓存的大小。
- query_cache_limit：单次查询缓冲区的大小。

通过增大内存来增大上面参数的值，也对提高性能有帮助。

6.5.2　增加应用缓存

上节的缓存是指 Linux 和 MySQL 的缓存，本节的缓存是指应用自身的缓存。

把应用的热数据存储在缓存中，如果缓存中有数据就不需要到数据库读取数据，从而达到提高性能的目的。

应用的缓存有以下两种。

- 本地缓存：把数据放在服务器的内存或文件中。
- 分布式缓存：通过使用分布式缓存工具 Redis 或 Memcache，可以缓存海量数据（Redis 的分布式需要借助第三方工具或者把 Redis 升级到 3.0 版本以上）。这两个软件的读写性能非常高，QPS（Query Per Second）能达到 1 万以上。如果项目中要考虑数据持久化，缓存可以选用 Redis；如果项目中不需要考虑数据持久化，选用 Redis 或 Memcache 都没问题。

6.5.3 用固态硬盘代替机械硬盘

用高性能的硬盘也能提高 MySQL 的性能。

现在电脑大多都在使用机械硬盘，其构造原理是硬盘里面由 1 张或几张可读写数据的存储盘体，盘体上有读写枪，有点像老式光碟机，硬盘里面还有马达带动存储盘转动，从而能读取到不同部分的数据。其优点是生产成本低、容量大，缺点是读写速度慢。

固态硬盘称为 SSD 硬盘，有点像平时的 U 盘，只是电路板更复杂，没有像机械硬盘那样的马达及存储碟盘，主要以半导体固体作为数据存储介质。其优点是速度快、稳定、寿命长，但价格比较高。

传统的硬盘中数据是存储在盘片，通过磁头移动和盘片转动的操作读取数据，但这造成了磁盘读写速度慢，越是不连续的文件，读写速度就越慢。对不连续的文件进行读写的操作称之为随机读写，实际上在日常使用中绝大多数硬盘读写操作都是随机类型，而 SSD 与传统硬盘的最大差异就在于随机读写速度，这就是由 SSD 的基本构造决定的。

SSD 硬盘最主要改变了数据存储和读取的方式，SSD 硬盘中数据存储在闪存芯片中，其是一种非易失性内存芯片，通过充电、放电的方式写入和擦除数据，速度相当快。读写操作中通过电路传输信号，因此也不会有传统硬盘的移动磁头和转动磁盘等操作，大大减少了处理时间。

机械硬盘的读取速度大概在 100MB/s 左右，而一般的 SSD 读取速度可达 400MB/s 甚至 600MB/s 以上，有些专业的 SSD 读取速度可达 4000MB/s 。

企业级固体硬盘厂商 Fusion-io 的产品能提供比一般的 SSD 硬盘更高的读写性能，IOPS（Input/Output Operations Per Second，即每秒进行读写（I/O）操作的次数）达到十万以上，其能使单机硬盘的读写性能得到一个量级的提升，更重要的是不需要添加服务器，减少运维上的成本。fusion-io 的产品提供了比一般 SSD 硬盘更高的性能，但价格也比一般 SSD 硬盘贵不少。

当后台使用了 SSD 硬盘后，可对 MySQL 做以下的优化措施。

- 日志和数据分开存储。MySQL 的日志是顺序读写的，不需要使用 SSD 这种随机读写性能高的设备，可以把 MySQL 的日志存储在机械硬盘。通过把顺序读写的数据和随机存储的数据分开存储，节省了 SSD 的存储空间。
- 调整了 MySQL 的参数。

```
//这个参数控制了 InnoDB 刷新日志和数据的模式，O_DIRECT 是告诉操作系统禁用缓存，然后使用 fsync () 的方式将数据刷入磁盘。fsync () 的方式确保数据已经被记录在硬盘上。
innodb_flush_method=O_DIRECT

//控制 MySQL 中一次刷新的脏页的数量，在使用 SSD 后其 io 能力大大增强，需要增大一次刷新脏页的数量。
innodb_io_capacity=10000
```

现阶段 SSD 硬盘的稳定性和可靠性都已经得到了大规模的验证，而且主流的云服务器提供商都已经支持 SSD 硬盘，强烈推荐使用！使用 SSD 硬盘是优化数据库性能最有效的手段。

6.5.4　SSD 硬盘+SATA 硬盘混合存储方案

FlashCache 是 Facebook 技术团队开源的 SSD 硬盘+SATA 硬盘混合存储方案，最初是为加速 MySQL 设计的。FlashCache 在文件系统和设备驱动之间增加一层缓存来实现对热数据的缓存。FlashCache 通过将传统硬盘上的热数据缓存在 SSD 硬盘，然后利用 SSD 硬盘优秀的读性能加速系统。这个方法和使用内存做缓存比较，虽然没有内存的读取速度快，但可用空间比内存大多了。

FlashCache 原理如图 6-2 所示。

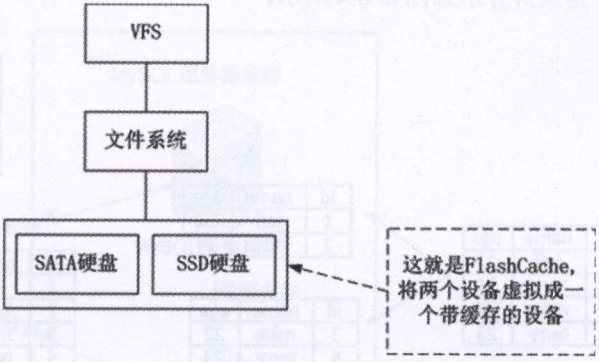


图 6-2　FlashCache 原理图

6.6　架构优化

当 MySQL 的性能瓶颈通过软件和硬件优化都无效后，开发人员就要考虑从架构上优化 MySQL，下面介绍 3 种从架构上优化 MySQL 的方法。

6.6.1 分表

当项目上线后，随着用户的增长，有些数据表的规模会以几何级增长，当数据达到一定规模后查询读取性能就下降很多，这时开发人员就要考虑分表。更新表数据时会更新索引，当单表数据量很大时这个过程比较耗时，这就是为什么对大表进行写操作会比较慢的原因，并且更新表数据会引起表锁或者行锁，这也会导致其他操作等待。

如果将大表拆分为多个子表，那么在更新或者查询数据的时候，压力会分散到不同的表上。由于分表之后每张表的数据较小，不管是查询还是更新的提高都得到极大的提升，即使出现最坏的“锁表”的情况，其他表还是可以继续使用。

分表有多种策略。

- 水平拆分：把一张表的数据分别保存在不同的表。
- 垂直拆分：把一张表的字段分别保存在不同的表。

水平拆分示意图如 6-3 所示。

垂直拆分示意图如 6-4 所示。

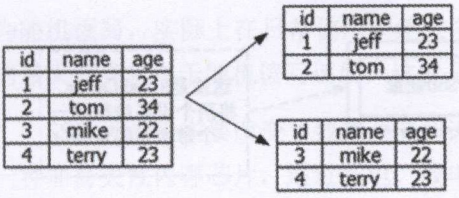


图 6-3　水平拆分示意图

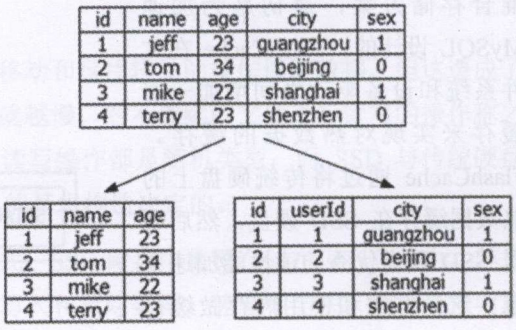


图 6-4　垂直拆分示意图

分表策略确定后还有一个必须考虑的问题：用户的数据都分散在不同的表中，之前的业务功能如何保证？比如说笔者要插入一条记录、更新一条记录、删除一条记录、查询统计数据，现在要怎么处理呢?

如果分表的存储引擎是 MyISAM，这里有一种很简单的处理方法。利用 MERGE 存储引擎将拆分的表合并成一张表。如果使用 InnoDB，也能通过 alter table 命令把 InnoDB 变为 MyISAM。

MERGE 存储引擎可以将 N 个子表联合在一起，看成是一个整表，实际上还是 N 个真实的子表。

这是早期可以采用的简单手动分表方式，当规模变大后，建议采用 “6.6.3 分库”中提到的分布式关系数据库产品。

6.6.2　读写分离

随着 App 的不断迭代和推广，数据不断增多，数据库的压力也越来越大，对 MySQL 的基础优化可能达不到最终的效果。针对这种情况，需要采取更高级的优化方法来优化数据库，数据库的读写分离策略就是其中的一种优化策略。读写分离的技术已经被应用于大量的 App 后台中。

读写分离是把对数据库的读和写操作分开对应于主/从数据库服务器，主数据库提供写操作，从数据库提供读操作（可以有多个从数据库）。因为大多数业务是以读为主，因此多个从数据库能有效减轻数据库的压力。

读写分离的架构如图 6-5 所示。

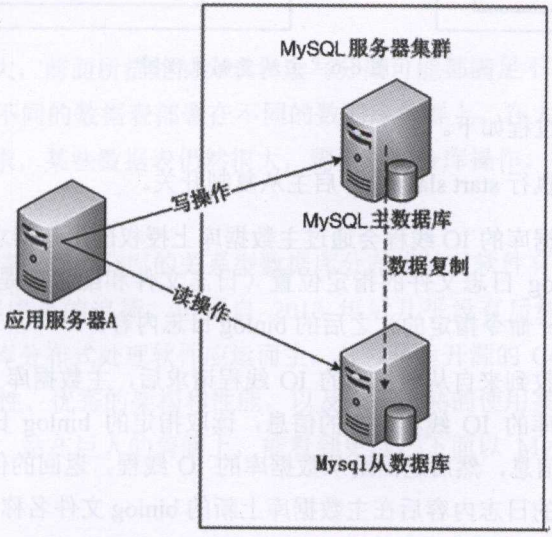


图 6-5　MySQL 读写分离架构

当主数据库进行写操作时，数据要同步到从数据库，这样才能有效保证数据库完整性。这称为数据库的主从复制。

MySQL 主从复制基于主服务器在二进制日志（binlog）中跟踪所有对数据库的更改实现。

因此要进行复制，必须在主服务器上启用二进制日志。

MySQL 主从复制是异步复制，在主数据与从数据库之间实现整个主从复制的过程有三个线程参与，其中两个线程（SQL 线程和 IO 线程）在从数据库，另外一个线程（IO 线程）在主数据。

主从复制的过程如图 6-6 所示。

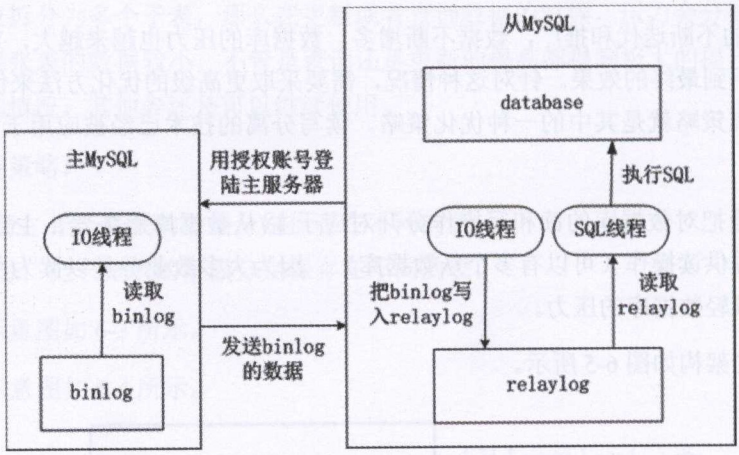


图 6-6　主从复制流程图

MySQL 主从复制过程如下。

（1）从数据库上执行 start slave，开启主从复制开关。

（2）此时，从数据库的 IO 线程会通过主数据库上授权的用户请求连接主数据库，并请求指定的 binlog 日志文件的指定位置（日志文件和配置都是配置主从服务器时执行 change master 命令指定的）之后的 binlog 日志内容。

（3）主数据库接收到来自从数据库的 IO 线程请求后，主数据库上负责复制的 IO 线程，根据从数据库的 IO 线程请求的信息，读取指定的 binlog 日志文件指定位置之后的 binlog 日志信息，然后返回给从数据库的 IO 线程。返回的信息除了日志内容外，还有本次返回的日志内容后在主数据库上新的 binlog 文件名称及在 binlog 中的位置。

（4）当从数据库上的 IO 线程获取来自主数据库上的 IO 线程发送日志内容及位置后，将 binlog 日志内容依次写入到从数据库自身的 relaylog（中继日志）文件（MySQL-relay-bin.XXXX）的最末端，并将新的 binlog 文件名和位置记录到 Master-info 文件中，以便下一次读取主数据库的新 binlog 日志时，能够告诉 master 服务器需要从新 binlog 日志的哪个文件哪个位置，开始请求新的 binlog 日志内容。

（5）从数据库的 SQL 线程会实时地检测本地 relay log 中新增加的日志内容，然后及时把 log 文件中的内容解析成在主数据库曾经执行的 SQL 语句，并在自身从数据库上按顺序执行这些 SQL 语句。

（6）经过了上面的过程，就可以确保在主数据库和从数据库执行同样的 SQL 语句。当复制状态正常的情况下，主数据库和从数据库的数据是一样的。

主从复制一直以来最受诟病的部分，就是主从复制的延迟问题，一般来说从数据库的压力比主数据库大多了，非常容易导入延迟。

延迟的解决方案如下。

- 首先定位延迟的瓶颈，如果是因为 IO 压力大，那么可以考虑采用升级硬件的方案，如把硬盘更换为 SSD 硬盘。
- 如果是因为单线程从 relaylog 中执行 MySQL 语句导致延迟，可以采用 MySQL 5.6 以上版本的多线程方案，或者采用 Tungsten 为代表的第三方并行复制工具。
- 如果以上的两个方案都不起作用，就要考虑采用下面介绍的分库策略。

6.6.3　分库

当数据规模不断增大，前面所描述的分表和读写分离可能都满足不了系统的性能需求，这时需要考虑分库，即把不同的数据表部署在不同的数据库集群上。在大型的 App 后台中，分表都有可能满足不了需求，某些数据表仍然很大，需要进行分库操作，即把一张表的数据分别存储在不同的数据库。

目前比较成熟的支持数据库分库的关系型数据库分布式处理软件有 Cobar。Cobar 自诞生之日起，就受到广大程序员的追捧，但是自 2013 年后几乎没有后续更新。在此情况下，MyCat 这个关系型数据库分布式处理软件应运而生，其以阿里开源的 Cobar 产品为基础研发，MyCat 的稳定性、可靠性、优秀的架构和性能，以及众多成熟的使用案例使得 MyCat 一开始就拥有一个很好的起点，站在巨人的肩膀上，能看到更远。下面以 MyCat 为例，讲解一下其基本的原理。

MyCat 以代理服务器的形式位于应用服务器与后台数据库之间，由于其是无状态，因此很容易部署 MyCat 集群实现负载均衡。对外开放的接口是 MySQL 通信协议，将应用服务器传过来的 SQL 语句按照路由的规则拆解转发到不同的后台数据库，并把结果汇总并返回。

MyCat 的部署模型如图 6-7 所示。

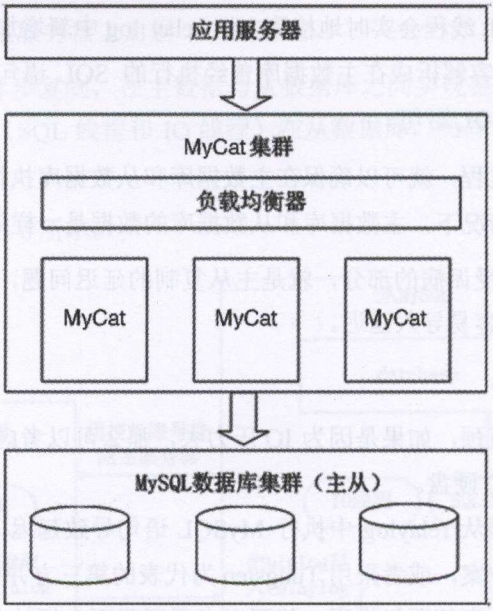


图 6-7　MyCat 的部署模型

MyCat 可以把一个逻辑上的数据库和数据表对应到物理上真实的数据库和数据表，用户只需要按照逻辑上的结构操作相关的数据就行，遮蔽了物理上的差异性。下面以图 6-8 所示的 MyCat 映射关系图来说明。

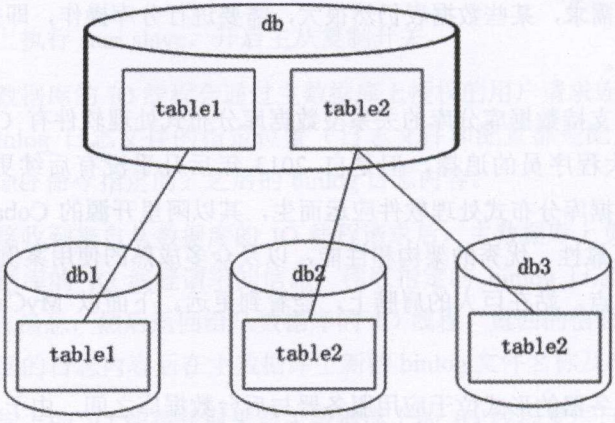


图 6-8　MyCat 映射关系图

MyCat 对外提供的数据库是 db，其中有两张表 table1 和 table2。

- table1 表的数据映射到物理数据库 db1 的 table1。

- table2 表的数据一部分映射到物理数据库 db2 的 table2，另一部分映射到物理数据库 db3 的 table2。

下面以一个实例来说明 MyCat 的工作流程，如图 6-9 所示。

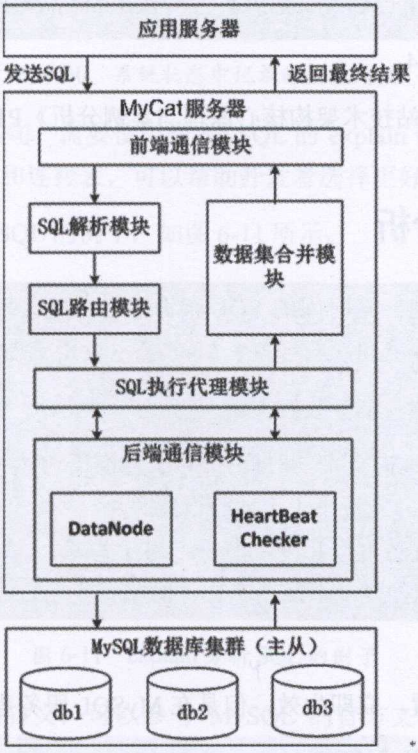


图 6-9　MyCat 的工作流程

Datanode 为 MyCat 的逻辑数据节点，映射到后端的某个物理数据库的 database。为了保证高可用，每个 Datanode 可配置多个引用地址（DataSource），当主 DataSource 被检测到不可用，MyCat 会切换到可用的 DataSource，这里的 DataSource 可认为是 MySQL 主从服务器的地址。

下面以一个例子讲解 MyCat 的工作流程：应用服务器向 MyCat 服务器发送 SQL 语句“select * from user where id in (30，31，32)”。MyCat 服务器的前端通信模块与应用服务器通信，前端通信模块收到这个 SQL 语句后交给 SQL 解析模块，SQL 解析模块解析完毕后根据查询条件（id in (30，31，32)）交给 SQL 路由模块，从路由规则得知，id 取摸余数为 0 的数据在 db1，id 取摸余数为 1 的数据在 db2，id 取摸余数为 2 的数据在 db3，于是把 SQL 拆解为“select * from user where id in (30)”，“select * from user where id in (31)”，“select * from user where id in (32)”转交给 SQL 执行模块执行，分别对应数据库 db1、db2、db3。SQL 执行模块

通过后端通信模块，分别在 db1、db2、db3 上执行相应的 SQL 语句。最后把数据库返回的结果通过数据集合并模块合并，返回给应用服务器。

参考资料：

（1）https://github.com/MyCATApache/MyCat-doc/blob/master/MyCat_In_Action_CN.doc《MyCat_In_Action_CN》P1-4

（2）李智慧著《大型网站技术架构核心原理与案例分析》P112-115

6.7 SQL 慢查询分析

SQL 慢查询是指执行超过一定时间的 SQL 查询语句，把这些 SQL 查询语句记录到慢查询日志，方便开发人员找出有性能问题的 SQL，针对这些 SQL 查询语句进行分析调优。

配置选项中慢查询相关的 3 个参数如下。

- long_query_time：定义慢查询的时间，SQL 查询语句执行时间大于该参数设置时间的 SQL 都会被记录下来，支持小于 1 秒的设置。
- slow_query_log：设置是否打开慢查询日志的开关。
- slow_query_log_file：设置慢查询日志文件的路径。

配置慢查询有两种方法。

方法一，通过命令行设置，立即生效，但是在 MySQL 服务重启后失效。

```
set global long_query_time=1;
set global slow_query_log=on;
set global slow_query_log_file='/data/slow.log';
```

方法二，在/etc/my.conf 中增加下面的配置选项，重启 MySQL 服务后生效。

```
[MySQLd]
long_query_time=1
slow_query_log=ON
slow_query_log_file=/data/slow.log
```

开发者也可使用 MySQL 自带的工具 mysqldumpslow 分析慢查询日志，例如，查看最慢的前 3 个 SQL 查询的命令格式如下。

```
mysqldumpslow -t 3 /data/slow.log
```

开启慢查询日志后可在系统状态中可看到共有多少个慢查询，如图 6-10 所示。

```
mysql> show global status like '%slow%';
+---------------------+-------+
| Variable_name       | Value |
+---------------------+-------+
| Slow_launch_threads | 0     |
| Slow_queries        | 0     |
+---------------------+-------+
2 rows in set (0.00 sec)
```

图 6-10　系统状态中记录的慢查询数

开发人员要分析慢查询语句，需要使用到 MySQL 的 explain 命令。explain 显示了 MySQL，如果使用索引处理 select 语句和连接表，可以帮助开发者选择更好的索引和更优化的 SQL 语句。

下面是使用 explain 分析 SQL 的例子，如图 6-11 所示。

```
mysql> explain select * from ek_region where id>2\G;
*************************** 1. row ***************
           id: 1
  select_type: SIMPLE
        table: ek_region
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3235
        Extra: Using where
1 row in set (0.00 sec)
```

图 6-11　explain 分析 SQL 的例子

关于 explain 输出结果的含义，可以参考 MySQL 的官方文档：http://dev.mysql.com/doc/refman/5.5/en/explain-output.html#explain_select_type。

6.8　云数据库简介

MySQL 数据库在大量的 App 后台广泛使用，由于其作为一个重要的基础组件，国内的云服务提供商纷纷推出了云数据库的产品。云数据库有大量的优点，笔者看重的是下面一些优点。

- 配置高性能的 SSD 硬盘。在“6.5.3 用固态硬盘代替机械硬盘”中介绍到，机械硬盘的读取速度大概在 100MB/s 左右，而一般的 SSD 读取速度可达 400MB/s 甚至 600MB/s 以上，有些专业的 SSD 读取速度可达 4000MB/s，数据库操作耗时中磁盘 IO 的耗时占了一个很大的比例，因此使用 SSD 硬盘能把性能提升一大截。
- 备份机制。每台云数据库拥有两个物理节点进行主从热备，主节点发生故障，快速换

至备节点。还有自动备份机制可以保存多天的备份数据以便于在灾难情况下进行数据恢复。这些措施都保证了 MySQL 的高可用。

- 完善的监控体系。在云数据库管理界面中有十多种性能资源（CPU，内存，磁盘，连接数，增删改查的 QPS，慢查询数目）监控视图，可对部分资源项设置阈值报警，并提供 WEB 操作、SQL 审计等多种日志。如果开发人员收到 App 用户反馈某段时期的服务有问题，这些历史数据能给开发人员排查问题提供很大的方便。另一方面，如果靠开发人员搭建监控服务获取这些数据，并把其显示为图表不知道要花多少时间，用云数据库很轻松就能实现图表显示数据的需求。
- 弹性扩展。开发人员可以根据数据库的实际负载情况升级硬件，从而获取更高的数据库性能。当 App 的访问量忽然爆发，系统性能已经不能满足用户的访问需求了，这时候解决爆发访问量最便捷的方法是升级硬件，而不是在代码层面优化性能。通过优化代码而提升性能不是短时间内可以完成的，而在云服务器上升级硬件，只需要几分钟甚至几十秒。

笔者一向倡导创业团队中架构原则是**“尽量使用成熟可靠的云服务和开源软件，自身只专注于业务逻辑”**。MySQL 作为一个成熟、稳定、通用的软件，云服务商已经提供了完善的基于 MySQL 的云数据库服务，使用云数据库不但节省了大量的运维成本，还能把自身的精力集中于业务逻辑。

6.9 灵活的存储结构

MySQL 是模式化结构，一张表中每行数据的字段是固定的。例如用户发的内容表初期可以用如表 6-1 所示的基本表结构。

表 6-1 内容表结构

id	user_id	content
10001111	1	“这是一条测试的内容”

随着业务的不断变化，这张表上可能添加更多的字段，例如发送这条内容时使用的设备信息等等，越来越多的字段需要放在这张表上，表结构的演进如表 6-2 所示。

表 6-2 内容表结构的演进

id	user_id	content	device_info	…
10001111	1	“这是一条测试的内容”	“华为荣耀 X4”	…

当表的数据量到了一定的程度（例如上千万，上亿），任何表结构的修改会对线上的业务产生巨大的影响，解决这个问题的方案就是把索引表和内容表分离。

索引表只存放需要索引的字段，保证高效的查询性能，索引表就只负责索引，不承担其他职责。索引表结构如表 6-3 所示。

表 6-3　索引表结构

id	content_id
1	10001111
2	10002222

内容表使用的是 kv 结构，k 是 feed_id，v 存储的是二进制数据，便于数据的变更，把关于这个表的其他内容存储在 v。v 为了便于扩展和节省存储空间，采用可扩展的序列化格式，例如 protocol buffer，内容表结构如表 6-4 所示。

表 6-4　内容表结构

content_id	content
10001111	[bytes]
10002222	[bytes]

6.10　故障排除案例

性能低下的查询引起的故障

故障现象：App 一旦使用搜索商家的功能，后台数据库 load 就居高不下，超过了正常水平。

原因分析：分析搜索商家的整个过程，商家的数据表有上百万的数据，搜索商家使用的 like 模糊查询， like 查询没有使用索引，每次查询都是遍历上百万的数据，性能的低下可想而知。

经验教训：对于这种大数据量的搜索，不应该为了快速开发而使用 MySQL 的模糊查询，使用 Sphinx 或者 Coreseek 等开源检索引擎实现这样的搜索要求。

第 7 章
Redis——App 后台高性能的缓存系统

用户对 App 后台的响应速度要求越来越高。把数据存储在硬盘上，由于受到硬盘读写速度的限制，因此数据读写的速度有限。内存比硬盘的读写速度高了一个数量级，因此基于内存的数据读取也比硬盘提高了一个数量级，而 Redis 正是基于内存存储数据，保证了数据快速的读写速度。同时 Redis 提供了丰富的数据类型（string、hash、list、set 及 zset），给开发者带来了很大的便利。

本章主要介绍 Redis 下面的内容。

- Redis 的简介
- 常用数据结构及应用场景
- 内存优化
- 集群
- 持久化
- 故障排除案例

7.1 Redis 简介

MySQL 被广泛用于数据存储，但 MySQL 读写速度慢，随着移动互联网的发展，越来越多的业务场景需要满足下面的需求。

- 少量数据需要被经常读写，同时对读写速度要求非常高。
- 能提供丰富的数据结构。
- 提供数据落地的功能。

如表 7-1 所示展示了计算机中主要存储介质的访问速度。

表 7-1　计算机中主要存储介质的访问速度

介质	时间
内存寻址	100 纳秒
从内存中顺序读取 1MB 的数据	250 000 纳秒
磁盘寻址	10 000 000 纳秒
从磁盘中顺序读取 1MB 的数据	20 000 000 纳秒

读者在表 7-1 中可了解，从内存中顺序读取 1MB 的数据耗时 250 000 纳秒，从磁盘中顺序读取 1MB 的数据耗时 20 000 000 纳秒，内存的读取速度是硬盘的读取速度的 80 倍。因此可以通过把内存数据存储在内存来提升系统的性能。

Redis 就是一个满足上面需求的开源 Key-Value 内存存储系统。Key-Value 存储系统在写入的时候，通过指定 Key 及其对应的 Value；读取的时候，通过指定 Key 就能读取 Value 的值。Redis 有以下的特点。

- 全部的数据操作在内存，保证了高速的读写速度。
- 提供丰富多样的数据类型：string、hash、list、set、sorted set、bitmap 和 hyperloglog。
- 提供了 AOF 和 RDB 两种数据的持久化方式，保证了 Redis 重启后数据不丢失。
- Redis 的所有操作都是原子性，同时 Redis 还支持对几个操作合并后的原子性操作，也即支持事务。

Redis 丰富多样的数据类型给数据存储提供了一种新的思路，其让广大的开发者存储数据的时候，不用再面对功能单一的数据库，开发者能利用 Redis 提供的灵活多变的数据结构和数据操作更巧妙地存储数据。

7.2　Redis 的常用数据结构及应用场景

下面介绍 Redis 5 种常用的数据结构（string、hash、list、set、sorted set），以及这些数据结构所适用的业务场景。

注意：为了便于读者的理解，本章中所描述的 Redis 数据模型在原版的基础上进行了一定程度上的简化，详细的模型请参阅书籍《Redis 设计与实现》（黄健宏著）。

7.2.1 string——存储简单的数据

1. 简介

string 类型是 Redis 中最基本的数据类型，其在 Redis 中是二进制安全，意味着这种数据类型可以接受任何格式的二进制数据，例如一张 JPEG 格式的图片或者 JSON 格式的字符串。在 Redis 中字符串类型最多可以容纳的数据长度是 512MB。

2. 数据模型

string 类型是基本的 Key-Value 结构，Key 可以看作某个数据在 Redis 中的唯一标识，Value 是具体的数据。表 7-2 展示了基本 Key-Value 数据模型：

表 7-2 基本 Key-Value 数据模型

Key	Value
“name”	“jeff”
“city”	“guangzhou”

在表 7-2 中，Key 为“name”，对应 Value 为“jeff”；Key 为“city”，对应 Value 为“Guagnzhou”。

3. 应用场景

由于 string 类型灵活，可以存储大量的数据，所以在 App 后台中，string 类型经常会用来缓存数据。例如 App 中常见的商品分类栏，这类界面的特点是：访问频率高，数据不经常变动（可能几天）。所以为了提高这个界面的访问速度，把这个界面的数据放在 Redis 的一个 Key-Value 结构中，一般情况下 App 后台就从这个 Key 读取数据；当这个界面的数据发生变化时，用新的数据覆盖这个 Key 的数据。

假设这个界面的数据对应的 Key 是“category”，Value 为这个界面的 JSON 数据，则 Redis 中对应的模型如表 7-3 所示。

表 7-3 分类界面对应的 Key-Value

Key	Value
“category”	{"常用分类":.....，"潮流女装":.....，...}

当 App 端需要通过 API 获取这个界面的 JSON 数据时，API 请求到达 App 后台通过 Redis 获取 Key“category”对应的值，命令如下。

```
get category
```

一般来说，App 端为了在网络不可用的时候也有良好的用户体验，会在 App 本地也缓存一

份数据，整个流程如图 7-1 所示。

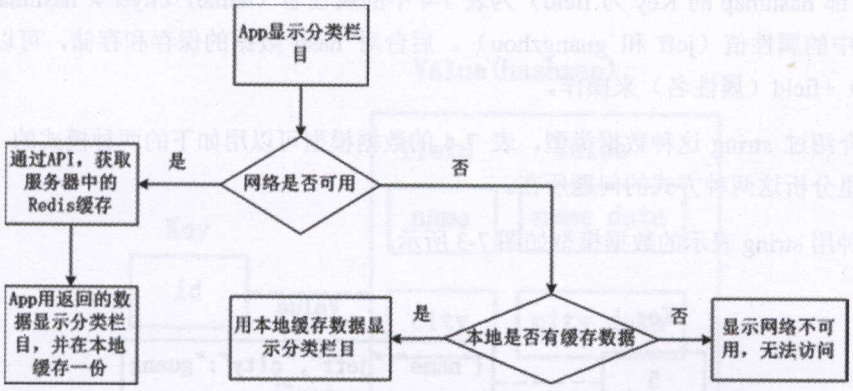


图 7-1　App 显示分类栏目流程图

7.2.2　hash——存储对象的数据

1. 简介

hash 类型很接近数据库模型，hash 的 Key 是个唯一值，Value 部分是个 hashmap 的结构。

2. 数据模型

在数据库中有这样一行用户数据，如表 7-4 所示。

表 7.4　用户数据

id	name	city
5	jeff	guangzhou

如果要在 Redis 中用 hash 结构存储，则数据模型如图 7-2 所示。

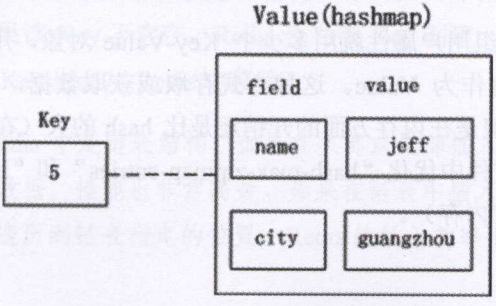


图 7-2　hash 数据模型

在这个 hash 数据模型中，Key 是用户 id 为 5，Value 是个 hashmap，hashmap 的 field（在 Redis 称内部 hashmap 的 Key 为 field）为表 7-4 中的属性名（name，city），hashmap 的 Value 为表 7-4 中的属性值（jeff 和 guangzhou）。后台对 hash 数据的保存和存储，可以通过 Key（用户 id）+field（属性名）来操作。

前面介绍过 string 这种数据类型，表 7-4 的数据模型可以用如下的两种模式的 string 表示，笔者在这里分析这两种方式的问题所在。

第一种用 string 表示的数据模型如图 7-3 所示。

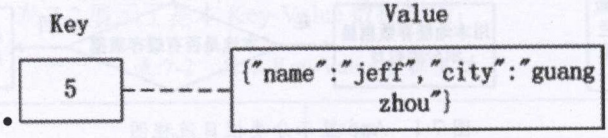


图 7-3　第一种用 string 表示的数据模型

第一种方式，Key 是用户的 id 为 5，Value 是一个 JSON 格式的字符串，这种方式的缺点是存储或获取 Value 时，把对象变为 JSON 格式或者把 JSON 格式变为对象需要额外的性能开销。另外如果开发者只需要修改 Value 中的 name 值，在这种格式中，开发人员必须要先获得 city 值，才能把其转化为符合 Value 格式的 JSON 值，增加了没必要的性能开销和复杂性。

第二种用 string 表示的数据模型如图 7-4 所示。

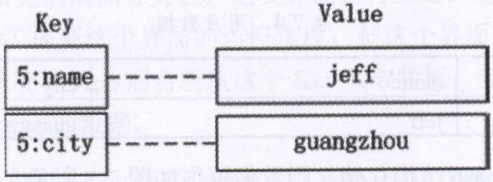


图 7-4　第二种用 string 表示的数据模型

第二种方式，有多少组用户属性就用多少个 Key-Value 对象，用户 id“5”加上对应的属性名来作为 Key，属性值作为 Value。这种方式存取或获取数据，虽然免去了如第一种方式 JSON/反 JSON 的开销，但是在内存方面的开销还是比 hash 的大（在下面内存优化的章节会描述，通过在 Redis 配置文件中优化“hash-max-zipmap-entries”和“hash-max-zipmap-value”这两个参数可以让 hash 更省内存）。

3. 应用场景

App 后台常见的功能是根据用户的 id 获取用户的信息。例如，根据用户的 id 获取用户的昵称、头像、所在地等信息。一般用户的信息是存储在数据库中，对于这种高频的数据访问，

不可能每次获取这些信息都读取数据库，自然而然开发人员会考虑到把用户的信息存储在 Redis 的 hash 中，如图 7-5 所示。

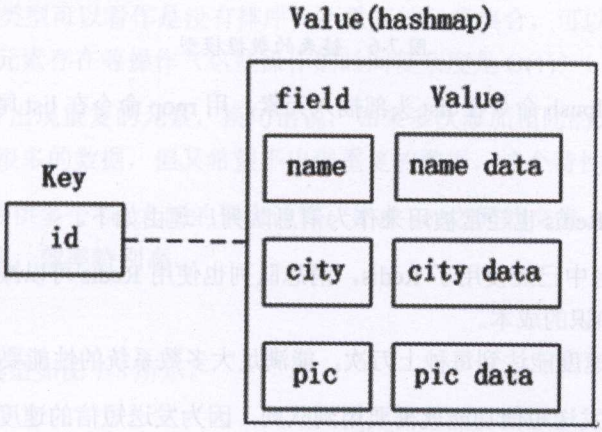


图 7-5　用户信息的 hash 数据

获得用户 id 后需要获取用户的数据，用 hgetall 命令获取 id 下所有的 field 和 value，命令如下：

```
hgetall id
```

注意：如果修改了数据库的用户数据，也要把这些数据同步更新到 Redis，用来防止 Redis 和数据库的数据不一致。

7.2.3　list——模拟队列操作

1. 简介

Redis 中 list 是按照插入顺序排序的字符串链表，可以在头部和尾部插入新的元素（即队列结构）。插入元素时如果该 Key 不存在，Redis 会为该 Key 创建一个新的链表，如果链表中所有的元素都被移除，该 Key 也会从 Redis 中移除。

注意：由于 list 在 Redis 中是链表结构，如果在头部或尾部插入新的元素，即使链表中存储了上百万的数据，性能也非常高效。如果在链表中插入元素，由于需要根据头部或尾部的指针遍历到链表指定的位置，Redis 的插入效率很低。

2. 数据模型

list 的数据模型如图 7-6 所示。

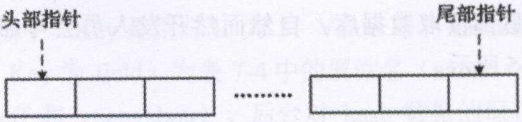


图 7-6　链表的数据模型

常见的操作是用 lpush 命令在 list 头部插入元素，用 rpop 命令在 list 尾取出数据。

3. 应用场景

在 App 后台中，Redis 也经常被用来作为消息队列，理由如下。

- 因为 App 后台中已经使用了 Redis，消息队列也使用 Redis 可以减少开发人员的维护成本和学习新知识的成本。
- Redis 的读写速度能达到每秒上万次，能满足大多数系统的性能要求。

App 后台常见的发送短信功能就需要用到队列，因为发送短信的速度慢，所以需要用到队列来实现异步操作（关于队列的知识，请查看“2.4 如何选择消息队列软件”这个章节），整个架构如图 7-7 所示。

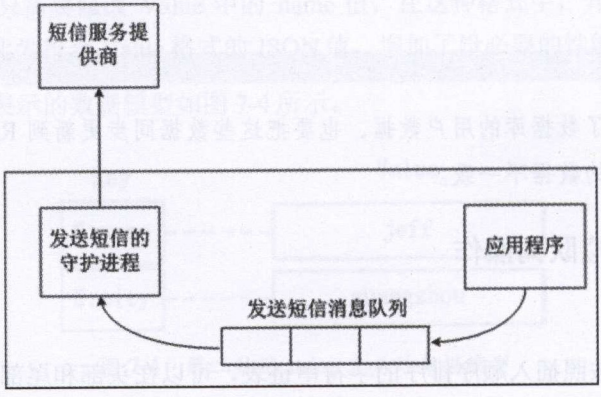


图 7-7　发送短信的架构图

发送短信的过程如下。

（1）应用程序把短信相关的信息（包括手机号、内容）转换为 JSON 字符串后放入“发送短信消息队列”。

（2）发送短信的守护进程是个在后台不断运行的程序，其不断地检测“发送短信消息队列”是否为空，如果不为空，就把信息从消息队列中取出。

（3）发送短信的守护进程把短信的内容发送短信平台的接口。

7.2.4　set——无序且不重复的元素集合

1. 简介

在 Redis 中 set 类型可以看作是没有排序、不重复的元素集合，可以在该类型上添加、删除元素或者判断某一元素存在等操作（这些操作的时间复杂度是 O(1)）。

set 集合中不允许出现重复的元素，换句话说，如果多次添加相同的元素，set 中只保留一份。当用户需要存储很多的数据，但又希望不出现重复的数据，这个特性就非常有用。

另外 set 类型还提供多个 set 之间的聚合计算，如求 set 之间的交集、差集或并集，这些操作是在 Redis 内部完成，效率特别高。

2. 数据模型

set 类型的数据模型如图 7-8 所示。

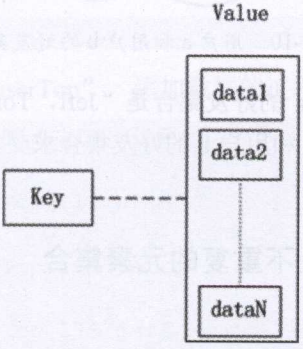


图 7-8　set 的数据模型

在图 7-8 中，set 类型的 Value 部分是一系列不重复的数据集合。

3. 应用场景

社交类型的 App 中，有的 App 当用户进入了一个用户的主页后会提示共同好友的信息，以方便用户扩展社交关系。提示共同好友的页面如图 7-9 所示。

图 7-9　提示共同好友

获取共同好友的算法如下：把用户 a 的所有好友取出来遍历，和用户 b 的所有好友一一比较，如果相同的话就是共同好友。

上面描述的算法其实就是求两个集合交集。在 Redis 的 set 类型的操作中已经包含了求交集的操作 sinter。如果把用户 a 的好友存储在集合 a 中，把用户 b 的好友存储在集合 b 中，通过求集合 a 和集合 b 的交集，就能获取用户 a 和用户 b 的共同好友，如图 7-10 所示。

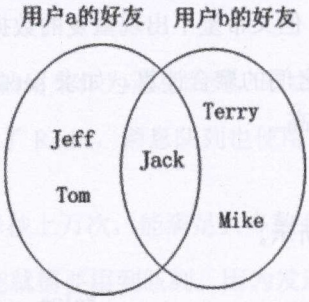


图 7-10　用户 a 和用户 b 的好友集合

在图 7-10 中，Redis 中用户 a 的好友集合是“Jeff，Tom，Jack”，用户 b 的好友集合是“Jack，Terry，Mike”，对用户 a 和用户 b 的好友集合求交集，就能得到用户 a 和用户 b 的共同好友是 Jack。

7.2.5　sorted set——有序且不重复的元素集合

1. 简介

sorted-set 类型与 set 类型非常相似，不允许出现重复的元素。其主要区别是 sorted-set 中提供了一个分数（score）与每一个成员对应，Redis 根据 score 对成员进行排序，而且插入是有序的，即插入后就自动排序。当 App 后台开发者需要有序且不重复的数据，选择 sorted-set 这种数据结构就非常合适。

需要特别注意：sorted-set 中的成员是不允许重复，但 score 是允许重复的。

2. 数据模型

sorted-set 的数据模型如图 7-11 所示。

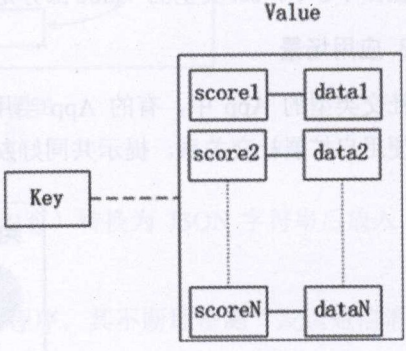


图 7-11　sorted-set 的数据模型

3. 应用场景

sorted-set 类型适用于各种类型的排行榜。如图 7-12 所示的用户人气榜。

排名	用户	好友数目
1	jeff	179
2	tom	127
3	mike	112
4	ekin	111
5	terry	104

图 7-12　用户人气榜

用户如果需要使用 sorted-set 实现如图 7-12 的用户人气榜，首先通过下面的命令把用户的数据添加到 Redis 中。

```
ZADD key score member
```

本例中 sorted-set 的 Key 是“userTop”，添加数据的过程如下所示。

```
127.0.0.1:6379>zadd userTop 112 mike
(integer) 1
127.0.0.1:6379>zadd userTop 111 ekin
(integer) 1
127.0.0.1:6379>zadd userTop 104 terry
(integer) 1
127.0.0.1:6379>zadd userTop 179 jeff
(integer) 1
127.0.0.1:6379>zadd userTop 127 tom
(integer) 1
```

接着通过下面的命令返回索引在 start 和 stop 之间的成员列表。

```
zrevrange key start stop [withscores]
```

其中 start 为 0 表示第一个成员，stop 为-1 表示最后一个成员，WITHSCORES 表示返回的结果中包含每个成员的分数），排序命令如下。

```
127.0.0.1:6379>zrevrange userTop 0 -1 WITHSCORES
 1) "jeff"
 2) "179"
 3) "tom"
 4) "127"
 5) "mike"
 6) "112"
```

```
 7) "ekin"
 8) "111"
 9) "terry"
10) "104"
```

上面的返回结果已按照分数从大到小排序了。

7.3 内存优化

由于 Redis 在内容中存储数据的特性，Reids 会占用大量的内存，Redis 的开发者也考虑到这一点，因此在 Redis 中提供了一系列的参数和方法来监控、控制和优化内存。

7.3.1 监控内存使用的状况

在通过 Redis 的终端命令 redis-cli 中输入命令“info”可查看 Redis 的各种统计信息，其中有关内存的统计信息如下。

```
# Memory
used_memory:12660096
used_memory_human:12.07M
used_memory_rss:14299136
used_memory_peak:15534680
used_memory_peak_human:14.82M
used_memory_lua:31744
mem_fragmentation_ratio:1.13
mem_allocator:jemalloc-3.2.0
```

在上面展示的参数中，3 个重要的内存统计信息的说明如下。

- used_memory_human : 以可读格式返回 Redis 分配的内存总量。
- used_memory_rss : 从操作系统的角度，返回 Redis 已分配的内存总量。这个值的结果，和 top 命令的输出一致。
- used_memory_peak_human : 以可读格式返回 Redis 的内存消耗峰值。

如果开发者在这里的统计数据中查看到内存使用过多，在不考虑使用 Redis 分布式存储的情况下，开发者务必要想办法优化 Redis 内存的使用情况。

7.3.2 优化存储结构

Redis 的开发者在配置文件中提供了一组参数来控制 hash、list、set、sorted-set 这些结构的内存存储方式。

在正常的情况下，hash 中的 value 是以 hashmap 的方式存储，如果 hashmap 的成员较少，或者 hashmap 的值的长度较少，会以类似于线性压缩表的方式（Redis 中称为 ziplist）的方式保存 hash 的数据，该控制参数对应于 Redis 配置文件中的下面两项：

```
hash-max-ziplist-entries 512
hash-max-ziplist-value 64
```

- hash-max-ziplist-entries：当 hashmap 内部的成员不超过 512 时，就采用 ziplist 的形式存储数据；当 hashmap 内部的成员超过 512 时，就采用 hashmap 的形式存储数据。
- hash-max-ziplist-value：当 hashmap 内部的成员的长度不超过 64 时，就采用 ziplist 的形式存储数据；当 hashmap 内部的成员的长度超过 64 时，就采用 hashmap 的形式存储数据。

注意：以上两个值任意一个超过了，hash 的存储方式就会转换为 hashmap。

当 hash 采用 ziplist 存储数据时，数据模型如下。

ZIPLISTENTRY HEAD	key1	value1	key2	value2	...	...	keyN	valueN	ZIPLISTENTRYEND

Redis 配置文件中下面这些参数的含义也是类似，分别控制 list 和 zset 是否采用 ziplist 的存储方式，set 是否使用 intset 的存储方式来节省内存。

```
list-max-ziplist-entries 512
list-max-ziplist-value 64
set-max-intset-entries 512
zset-max-ziplist-entries 128
zset-max-ziplist-value 64
```

当 list 采用 ziplist 存储数据时，数据模型如下。

ZIPLISTENTRY HEAD	data1	data2	...	...	keyN	ZIPLISTENTRYEND

当 zset 采用 ziplist 存储数据时，数据模型如下。

ZIPLISTENTRY HEAD	data1	score1	data2	score2	...	...	ZIPLISTENTRYEND

注意：set 使用了 intset 的结构来节省内存，intset 数据模型如图 7-13 所示。

上面的这些参数值不是设置得越大越好，例如，hash 的数据结构中如果用 hashmap 存储数据，查找和操作的时间复杂度都是 O(1)，采用了 ziplist 后，由于 ziplist 是个线性表结构，查找和操作的时间复杂度会变成 O(n)。如果数据成员量不大，则影响不大，当数据成员量变大后，则会严重影响性能。开发者需要在时间和空间之间认真衡量怎么设置上面所述的参数。

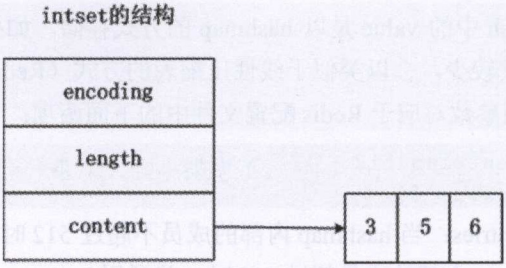


图 7-13　intset 的数据模型

7.3.3　限制使用的最大内存

如果 Redis 因为使用了过多的物理内存而导致使用交换分区后，很容易导致 Redis 崩溃。为了防止 Redis 使用过多的物理内存，可以通过配置文件中“maxmemory”的参数限制 Redis 使用的物理内存。

当 Redis 使用的物理内存达到了限制值，任何 write 操作（比如 set）会触发“数据清除策略”，通过配置文件中的“maxmemory-policy”来采用特定的“数据清除策略”，Redis 中定义的数据清除策略如下。

- volatile-lru：对设置了过期时间的数据，将过期的数据移除，或者按照 LRU（先进先出）算法移除。如果移除后的空闲内存还不满足写入数据所需的内存空间，则提示写入异常。
- allkeys-lru：对所有的数据采用 LRU（先进先出）算法。
- volatile-random:：对设置了过期时间的数据，采取“随机选取”算法移除数据。如果移除后的空闲内存还不满足写入数据所需的内存空间，则提示写入异常。
- allkeys-random：对所有的数据采取“随机选取”算法移除数据，直到空闲内存满足写入数据所需的内存空间为止。
- volatile-ttl：对设置了过期时间的数据采取 TTL 算法(最小存活时间)，移除即将过期的数据。
- noeviction：不做任何干扰操作，直接返回写入异常。

7.3.4　设置过期时间

Redis 中可以通过下面的命令设置 Key 的超时时间。

```
EXPIRE key seconds
```

超过超时时间后，该 Key 与对应的数据会被 Redis 删除。通过删除过期的 Key，可以在一

定程度上优化内存的使用。

当设置了超时时间的数据被修改后，设置的超时时间会失效。

在 Redis 的每个数据库中（Redis 有 16 个 db，默认是使用 db0），会使用下面的数据模型记录下所有设置了过期时间的 Key 和过期的时间（用时间戳表示，时间戳精确到毫秒），如图 7-14 所示。

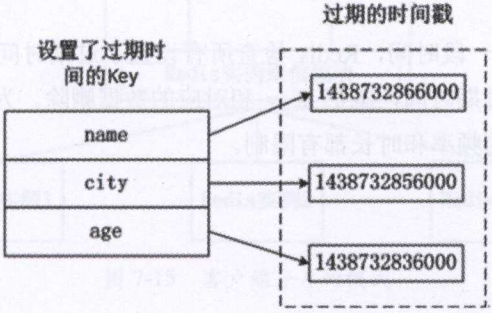


图 7-14　设置了过期的 Key

Redis 用如下的步骤检查某个 Key 是否过期。

（1）检查 Key 是否存在于设置了过期时间的 Key 中，如果存在，则取出过期时间。

（2）检查当前时间的时间戳是否大于 Key 的过期时间，如果是的话，则表示 Key 已过期，否则，Key 未过期。

设置了 Key 的过期时间后，Redis 采用下面的两种策略删除过期的 Key。

1. 惰性删除

Redis 操作 Key 时，如果发现 Key 已经过期了，则删除过期的 Key。

这种策略的好处是不占用过多的 CPU 资源，只有 Redis 操作 Key 时才检查，保证不会在其他 Key 上消耗 CPU 资源。

坏处是只有操作 Key 时才检查该 Key 是否过期，这样过期的 Key 的数据依然长期存储在内存中，占据内存的空间。

使用惰性删除策略时，如果内存中存在大量的过期的 Key，而这些 Key 没有被访问过会占用大量的内存空间，操作系统无法释放内存。这种删除策略对于数据都存储于内存的 Redis 来说非常糟糕。

如果 App 后台把 Redis 作为一个存储系统，App 业务当中肯定会存储一些冷数据，例如一

些不活跃的用户数据，这些用户注册后不再打开 App，就变成冷数据。这些冷数据的特点是当写入后很长时间内都不会被访问。如果只依赖于 Redis 的惰性删除，这部分冷数据一直占用着内存，没法清理内存空间。

2. 定期删除

Redis 为了补救惰性删除策略的不足，释放更多的内存，也对过期的 Key 同时采用了定期删除的策略。

定期删除策略是每隔一段时间，Redis 检查所有设置了过期时间的 Key，如果发现当前时间已经超过了该 Key 的过期时间，就把 Key 和对应的数据删除。为了保证 Redis 的高性能，Redis 执行定期删除策略的频率和时长都有限制。

7.4 集群

由于 Redis 出众的性能，其在众多的移动互联网企业中得到广泛的应用。Redis 在 3.0 版本前只支持单实例模式，虽然现在的服务器内存可以达到 100GB、200GB 的规模，但是单实例模式限制了 Redis 没法满足业务的需求（例如新浪微博就曾经在用 Redis 存储了超过 1TB 的数据）。Redis 的开发者 antirez 早在博客上提出在 Redis3.0 版本中加入集群的功能，但 3.0 版本等到 2015 才发布正式版。各大企业在 3.0 版本还没发布前为了解决 Redis 的存储瓶颈，纷纷推出了各自的 Redis 集群方案。这些方案的核心思想是把数据分片（sharding）存储在多个 Redis 实例中，每一片就是一个 Redis 的实例。下面介绍 Redis 的集群方案。

7.4.1 客户端分片

客户端分片是把分片的逻辑放在 Redis 客户端实现，通过 Redis 客户端预先定义好的路由规则，把对 Key 的访问转发到不同的 Redis 的实例中，最后把返回结果汇集。这种方案的模式如图 7-15 所示。

客户端分片的好处是所有的逻辑都是可控，不依赖于第三方分布式中间件。开发人员清楚怎么实现分片、路由的规则，不用担心踩坑。

客户端分片的方案有下面的坏处。

- 这是一种静态的分片方案，需要增加或者减少 Redis 实例的数量，都需要手工调整分片的程序。

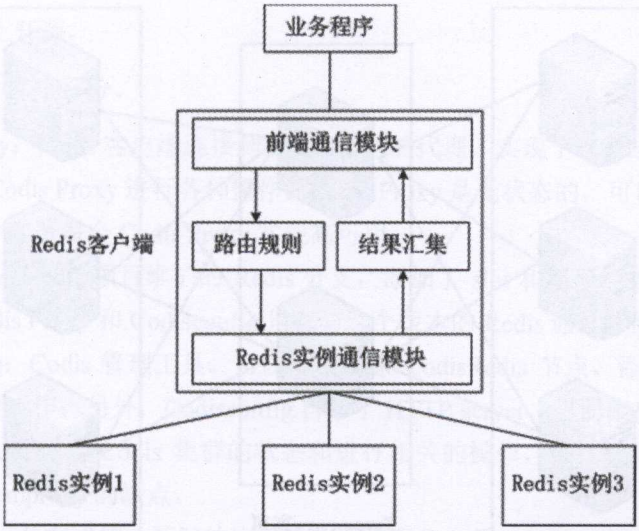


图 7-15　客户端分片的模式

- 可运维性差，集群的数据出了任何问题都需要运维人员（查看 Redis 实例的部分）和开发人员（查看客户端的分片逻辑部分）一起合作，延迟了解决问题的速度，增加了跨部门沟通的成本。
- 在不同的客户端程序中，维护相同的分片逻辑成本巨大。例如，系统中有两套业务系统共用一套 Redis 集群，一套业务系统用 Java 实现，另一套业务系统用 PHP 实现。为了保证分片逻辑的一致性，在 Java 客户端中实现的分片逻辑也需要在 PHP 客户端实现一次。把相同的逻辑在不同的系统中分别实现，这种设计本来就非常糟糕，而且需要耗费巨大的开发成本保证两套业务系统分片逻辑的一致性。

7.4.2　Twemproxy

Twemproxy 是由 Twitter 开源的 Redis 代理，其基本原理是：Redis 客户端把请求发送到 Twemproxy，Twemproxy 根据路由规则发送到正确的 Redis 实例，最后 Twemproxy 把结果汇集返回给客户端。

Twemproxy 通过引入了一个代理层，将多个 Redis 实例进行统一管理，使 Redis 客户端只需要在 Twemproxy 上进行操作，而不需要关心后面有多少个 Redis 实例，从而实现了 Redis 的集群。

Twemproxy 集群架构如图 7-16 所示。

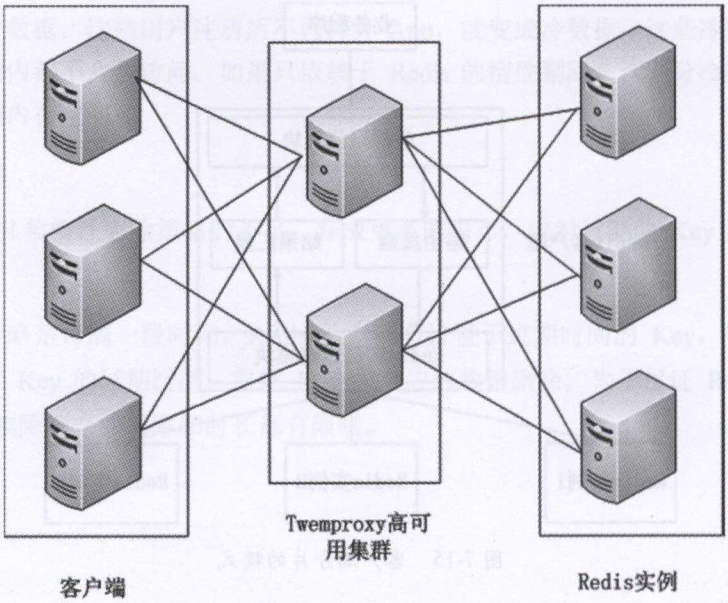


图 7-16　Twemproxy 集群架构

Twemproxy 的优点如下。

- 客户端像连接 Redis 实例一样连接 Twemproxy，不需要改任何的代码逻辑。
- 支持无效 Redis 实例的自动删除。
- Twemproxy 与 Redis 实例保持连接，减少了客户端与 Redis 实例的连接数。

Twemproxy 有如下不足。

- 由于 Redis 客户端的每个请求都经过 Twemproxy 代理才能到达 Redis 服务器，这个过程中会产生性能损失。
- 没有友好的监控管理后台界面，不利于运维监控。
- 最大的问题是 Twemproxy 无法平滑地增加 Redis 实例。对于运维人员来说，当因为业务需要增加 Redis 实例时工作量非常大。

Twemproxy 作为最被广泛使用、最久经考验、稳定性最高的 Redis 代理，在业界被广泛使用。

7.4.3　Codis

Twemproxy 不能平滑增加 Redis 实例的问题带来了很大的不便，于是豌豆荚自主研发了 Codis，一个支持平滑增加 Redis 实例的 Redis 代理软件，其基于 Go 和 C 语言开发，并于 2014

年 11 月在 Github 上开源。

Codis 包含下面 4 个部分。

- Codis Proxy：Redis 客户端连接到 Redis 实例的代理，实现了 Redis 的协议，Redis 客户端连接到 Codis Proxy 进行各种操作。Codis Proxy 是无状态的，可以用 Keepalived 等负载均衡软件部署多个 Codis Proxy 实现高可用。
- CodisRedis：Codis 项目维护的 Redis 分支，添加了 slot 和原子的数据迁移命令。Codis 上层的 Codis Proxy 和 Codisconfig 只有与这个版本的 Redis 通信才能正常运行。
- Codisconfig：Codis 管理工具。可以添加删除 CodisRedis 节点，添加删除 Codis Proxy，数据迁移等操作。另外，Codisconfig 自带了 HTTP server，里面集成了一个管理界面，方便运维人员观察 Codis 集群的状态和进行相关的操作，极大提高了运维的方便性，弥补了 Twemproxy 的缺点。
- ZooKeeper：分布式的、开源的应用程序协调服务，是 Hadoop 和 Hbase 的重要组件，其为分布式应用提供一致性服务，提供的功能包括：配置维护、名字服务、分布式同步、组服务等。Codis 依赖于 ZooKeeper 存储数据路由表的信息和 Codis Proxy 节点的元信息。另外，Codisconfig 发起的命令都会通过 ZooKeeper 同步到 Codis Proxy 的节点。

Codis 的架构如图 7-17 所示。

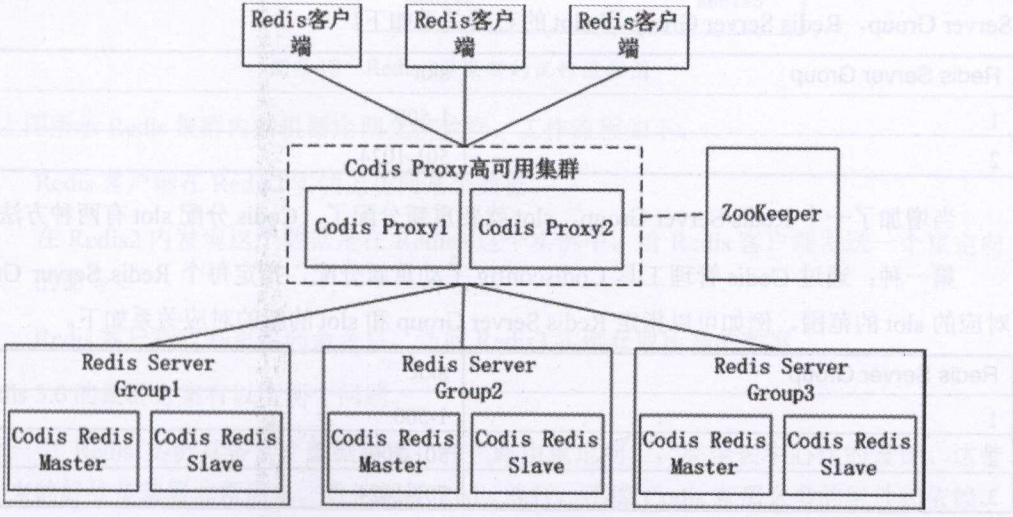


图 7-17　Codis 的架构图

在图 7-17 的 Codis 的架构图中，Codis 引入了 Redis Server Group，其通过指定了一个主

CodisRedis 和一个或多个从 CodisRedis，实现了 Redis 集群的高可用。当一个主 CodisRedis 挂掉时，Codis 不会自动把一个从 CodisRedis 提升为主 CodisRedis，这涉及到数据的一致性问题（Redis 本身的数据同步是采用主从异步复制，当数据在主 CodisRedis 写入成功时，从 CodisRedis 是否已读入这个数据是没法保证的），需要管理员在管理界面上手动把从 CodisRedis 提升为主 CodisRedis。

如果觉得麻烦，豌豆荚也提供了一个工具 Codis-ha，这个工具会在检测到主 CodisRedis 挂掉的时候将其下线并提升一个从 CodisRedis 为主 CodisRedis。

Codis 中采用预分片的形式，启动的时候就创建了 1024 个 slot，1 个 slot 相当于 1 个箱子，每个箱子有一个固定的编号，范围是 1~1024。slot 这个箱子用作存放 Key，至于 Key 存放到哪个箱子，可以通过算法“crc32(key)%1024”获得一个数字，这个数字的范围一定是 1~1024 之间，Key 就放到这个数字对应的 slot。例如，如果某个 Key 通过算法“crc32(key)%1024”得到的数字是 5，就放到编码为 5 的 slot（箱子）。1 个 slot 只能放 1 个 Redis Server Group，不能把 1 个 slot 放到多个 Redis Server Group 中。1 个 Redis Server Group 最少可以存放 1 个 slot，最大可以存放 1024 个 slot。因此，Codis 中最多可以指定 1024 个 Redis Server Group。

Codis 最大的优势在于支持平滑增加（减少）Redis Server Group（Redis 实例），能安全、透明地迁移数据，这也是 Codis 有别于 Twemproxy 等静态的分布式 Redis 解决方案的地方。Codis 增加了 Redis Server Group 后，就牵涉到 slot 的迁移问题。例如，系统有两个 Redis Server Group，Redis Server Group 和 slot 的对应关系如下。

Redis Server Group	slot
1	1-500
2	501-1024

当增加了一个 Redis Server Group，slot 就要重新分配了。Codis 分配 slot 有两种方法。

第一种：通过 Codis 管理工具 Codisconfig 手动重新分配，指定每个 Redis Server Group 所对应的 slot 的范围，例如可以指定 Redis Server Group 和 slot 的新的对应关系如下。

Redis Server Group	slot
1	1-500
2	501-700
3	701-1024

第二种：通过 Codis 管理工具 Codisconfig 的 rebalance 功能，会自动根据每个 Redis Server Group 的内存对 slot 进行迁移，以实现数据的均衡。

7.4.4　Redis 3.0 集群

Redis 3.0 集群采用了 P2P 的模式，完全去中心化。Redis 把所有的 Key 分成了 16384 个 slot，每个 Redis 实例负责其中一部分 slot。集群中的所有信息（节点、端口、slot 等），都通过节点之间定期的数据交换而更新。

Redis 客户端在任意一个 Redis 实例发出请求，如果所需数据不在该实例中，通过重定向命令引导客户端访问所需的实例。

Redis 3.0 集群的工作流程如图 7-18 所示。

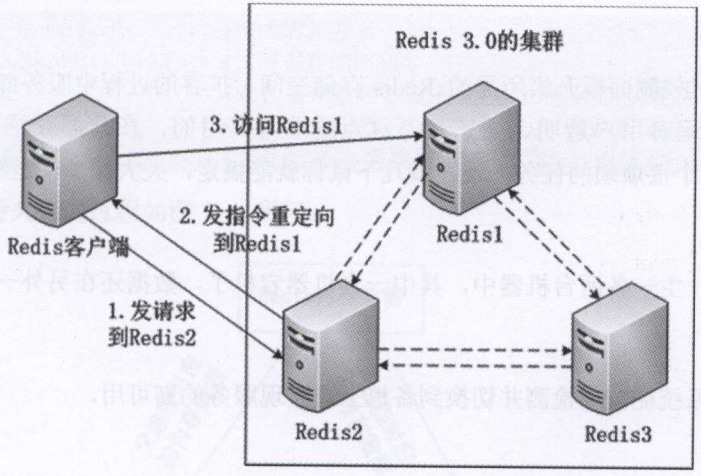


图 7-18　Redis 3.0 集群的工作流程图

如上图所示 Redis 集群内的机器定期交换数据，工作流程如下。

1. Redis 客户端在 Redis2 实例上访问某个数据。
2. 在 Redis2 内发现这个数据是在 Redis3 这个实例中，给 Redis 客户端发送一个重定向的命令。
3. Redis 客户端收到重定向命令后，访问 Redis3 实例获取所需的数据。

Redis 3.0 的集群方案有以下两个问题。

- 一个 Redis 实例具备了“数据存储”和“路由重定向”，完全去中心化的设计。这带来的好处是部署非常简单，直接部署 Redis 就行，不像 Codis 有那么多的组件和依赖。但带来的问题是很难对业务进行无痛的升级，如果哪天 Redis 集群出了什么严重的 Bug，就只能回滚整个 Redis 集群。

- 对协议进行了较大的修改，对应的 Redis 客户端也需要升级。升级 Redis 客户端后谁能确保没有 Bug？而且对于线上已经大规模运行的业务，升级代码中的 Redis 客户端也是一个很麻烦的事情。

综合上面所述的两个问题，Redis 3.0 集群在业界并没有被大规模使用。

7.4.5 云服务器上的集群服务

国内的云服务器提供商阿里云、UCloud 等均推出了基于 Redis 的云存储服务。这个服务的特性如下。

1. 动态扩容

用户可以通过控制面板升级所需的 Redis 存储空间，扩容的过程中服务部不需要中断或停止，整个扩容过程对用户透明、无感知，这点是非常实用的，在前面介绍的方案中，解决 Redis 平滑扩容是个很烦琐的任务，现在按几下鼠标就能搞定，大大减少了运维的负担。

2. 数据多备

数据保存在一主一备两台机器中，其中一台机器宕机了，数据还在另外一台机器上有备份。

3. 自动容灾

主机宕机后系统能自动检测并切换到备机上，实现服务的高可用。

4. 实惠

很多情况下为了使 Redis 的性能更高，需要购买一台专门的服务器用于 Redis 的存储服务，但这样子 CPU、内存等资源就浪费了，购买 Redis 云存储服务就很好地解决了这个问题。

有了 Redis 云存储服务，能使 App 后台开发人员从烦琐运维中解放出来。App 后台要搭建一个高可用、高性能的 Redis 服务，需要投入相当的运维成本和精力。如果使用云存储服务，就没必要投入这些成本和精力，可以让 App 后台开发人员更专注于业务。

7.5 持久化

Redis 是一个支持持久化操作的内存数据库，通过持久化机制把内存中的数据保存在硬盘文件。当 Redis 重启后通过把硬盘文件重新加载到内存，就能达到恢复数据的目的。

Redis 常用的持久化机制有下面两种。

- RDB

- AOF

下面为读者详细讲述这两种持久化机制。

7.5.1　RDB

RDB 是 Redis 默认的持久化方式，这种方式是按照一定的时间周期策略把内存的数据以快照的形式写入到硬盘的二进制文件。RDB 默认的数据文件是 dump.rdb，该数据文件能在配置文件中修改。

下面是 Redis 配置文件中有关 RDB 的主要参数。

```
dbfilename dump.rdb          #快照的文件名
dir /var/lib/redis/6379      #快照保存的路径
save 900 1                   #当有 1 个数据被改变时，900 秒刷新到硬盘一次
save 300 10                  #当有 10 个数据被改变时，300 秒刷新到硬盘一次
save 60 10000                #当有 10000 数据被改变时，60 秒刷新到硬盘一次
```

执行 RDB 持久化的过程如图 7-19 所示。

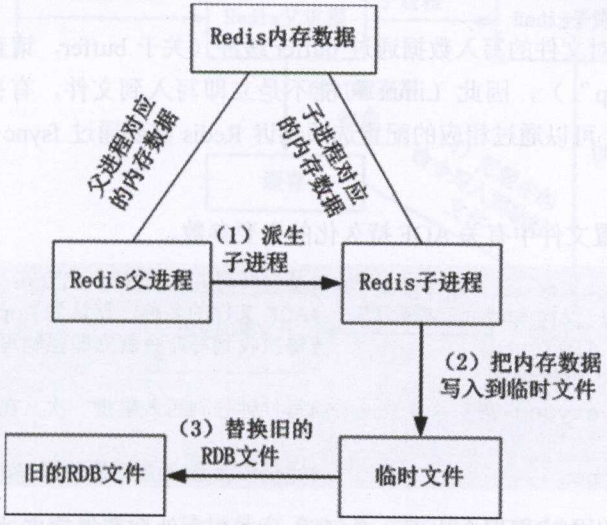


图 7-19　RDB 持久化的过程

在图 7-19 中，RDB 持久化的过程如下。

1. 根据配置文件中执行 RDB 的时机，Redis 调用 fork 生成子进程，这样就有了 Redis 的子进程和父进程。

2. 父进程继续处理客户端发送的请求，子进程把其内存的数据写入到临时文件。由于

Linux 操作系统的特性，父进程和子进程会共享相同的内存空间，所以子进程的数据是和 fork 时 Redis 中内存的数据一样的。

3. 子进程写入临时文件完毕后，用临时文件替换 RDB 的数据文件，子进程退出。

需要注意的是，每次持久化的过程都是把 Redis 内存数据完整地写入到磁盘，并不是只写入修改的数据，因此，如果 Redis 内存数据量大，那么就会造成频繁的写入操作，可能会严重影响性能。

由于 RDB 的方式是每隔一段时间才把内存数据持久化，如果 Redis 意外退出会丢失最后一次持久化后的所有数据。为了防止这个问题，可以采用下面介绍的另外一种持久化方式——AOF。

7.5.2 AOF

使用 AOF 的持久化方式，Redis 会把每个写入命令通过 write 函数追加到持久化文件中（默认文件是 Appendonly.aof），当 Redis 重启的时候会通过执行持久化文件的写命令重建内存数据。

由于 Linux 会把对文件的写入数据通过 buffer 缓冲（关于 buffer，请查阅“4.2.1 全面了解系统资源情况——top”），因此 Linux 可能不是立即写入到文件，有丢失数据的风险。在 Redis 的配置文件中，可以通过相应的配置选项告诉 Redis 需要通过 fsync 函数强制 Linux 写入到磁盘的时机。

下面是 Redis 配置文件中有关 AOF 持久化的主要参数。

```
    Appendonly no                          #是否开启 AOF 的持久化方式
    Appendfilename "Appendonly.aof"        #AOF 文件的名称，默认为 Appendonly.aof
    # Appendfsync always                   #每次收到写命令就立即强制写入到磁盘，能保证完全
持久化，但速度也最慢，不推荐
    Appendfsync everysec                   #每秒钟强制写入磁盘一次，在性能和持久化方面做了
很好的折中，推荐
    # Appendfsync no                       #完全依赖 Linux，性能最好，但持久化没保证
```

用 AOF 的持久化方式慢慢会出现一个问题：AOF 文件会变得越来越大。例如，有一个写命令“set num 1”，然后执行了 100 次写命令“incr num”，这时 num 的值为 101，这 100 次“incr”操作都会记录到持久化文件，但重建内存数据时，实际只需要执行“set num 101”就可以了，无须先执行“set num 1”再执行 100 次“incr num”。

为了压缩 AOF 文件，Redis 提供了 bgrewriteaof 命令，Redis 收到这个命令后会以类似创建 RDB 文件的方式将内存数据以命令的形式保存到临时文件中，最后替换原文件。

下面是 Redis 配置文件中有关 bgrewriteaof 命令的主要参数。

```
no-Appendfsync-on-rewrite yes      #在日志重写时，不进行命令追加，而将其放在缓冲区中
auto-aof-rewrite-percentage 100    #当前 AOF 文件大小是上次日志重写的 AOF 文件大小
的二倍时，自动启动新的日志重写过程。
auto-aof-rewrite-min-size 64mb     #当前 AOF 文件重写的最少值
```

当“auto-aof-rewrite-percentage”和“auto-aof-rewrite-min-size”这两个条件都满足时，才会触发 bgrewriteaof 命令。

执行 bgrewriteaof 命令过程如图 7-20 所示。

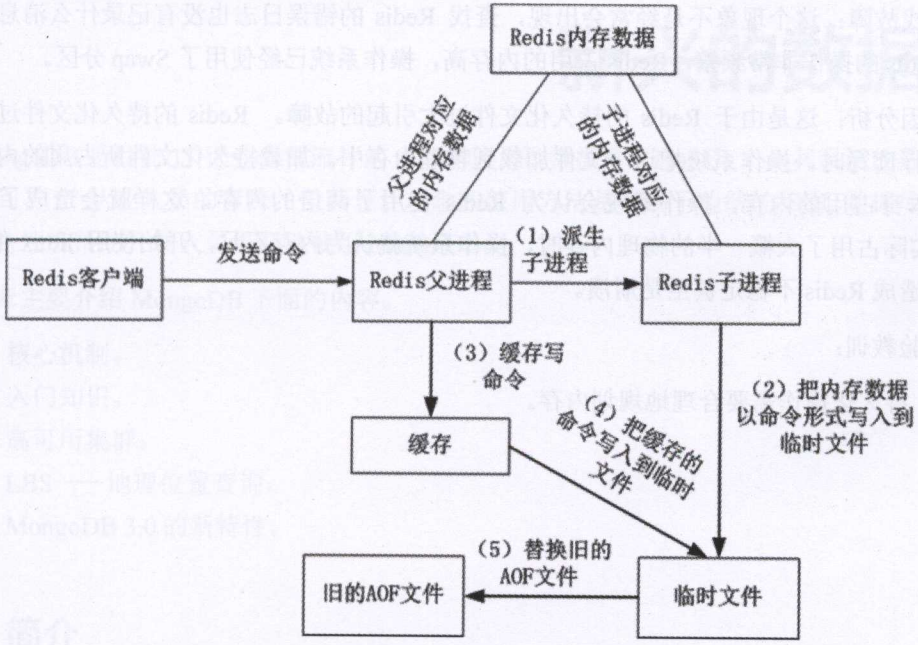


图 7-20　执行 bgrewriteaof 命令过程

图 7-20 执行 bgrewriteaof 命令过程如下。

1. Redis 调用 fork 生成子进程，这样就有了 Redis 的子进程和父进程。

2. 父进程继续处理客户端发送的请求，子进程把内存数据以命令的形式写入到临时文件。由于 Linux 操作系统的特性，父进程和子进程会共享相同的内存空间，所以子进程的数据是和 fork 时 Redis 中内存的数据一致的。

3. 在子进程写临时文件的过程中，父进程把收到的写命令缓存起来。

4. 子进程写入临时文件完毕，子进程通知父进程，父进程把缓存中的写入命令追加到临时

文件。

5. 临时文件替换 AOF 文件，父进程继续把新增的写命令追加到 AOF 文件，子进程退出。

7.6 故障排除案例

Redis 崩溃的故障

故障现象：在测试某个业务的过程中，发现 Redis 偶尔会崩溃。

查找故障：这个现象不是经常会出现，查找 Redis 的错误日志也没有记录什么消息。该业务写 Redis 的操作非常频繁，Redis 占用的内存高，操作系统已经使用了 Swap 分区。

原因分析：这是由于 Redis 的持久化文件过大引起的故障。 Redis 的持久化文件过大并要对其进行读写时，操作系统把这个文件加载到物理内存中。加载持久化文件所占用的内存加上 Redis 本身占用的内存，操作系统会认为 Redis 使用了两倍的内存。这样就会造成了当如果 Redis 实际占用了大概一半的物理内存时，操作系统就认为内存不足，开始使用 liunx 的 Swap 分区，造成 Redis 不稳定甚至是崩溃。

经验教训：

- 持久化操作需要合理地规划内存。

第 8 章
MongoDB——App 后台新兴的数据库

MongoDB 是目前 IT 行业非常流行的一种非关系型数据库（NoSQL），其灵活的数据存储方式得到了 IT 从业人员的青睐。但其设计理念和使用方法是有别于传统的 SQL 数据库的，学习这些特性将帮助开发人员更好地使用 MongoDB。

本章主要介绍 MongoDB 下面的内容。

- 核心机制。
- 入门知识。
- 高可用集群。
- LBS——地理位置查询。
- MongoDB 3.0 的新特性。

8.1 简介

MongoDB 是一个介于关系型数据库和非关系型数据库之间的产品，是非关系型数据库当中功能最丰富、最像关系型数据库的数据库。其是由 10gen 公司基于 C++语言编写，旨在提供可扩展的高性能数据存储解决方案。知名的 IT 公司中使用 MongoDB 来构建自己的核心应用有 Foursquare、eBay、Cisco、MetLife、Adobe 等，国内也有众多的团队将 MongoDB 作为首选数据库。

MongoDB 支持的数据结构非常松散，数据采用 bson 格式，可以存储比较复杂的数据类型。bson 是由 10gen 开发的一个数据格式，目前主要用于 MongoDB 中，是 MongoDB 的数据存储格式。

MongoDB 的主要特点如下。

- 读写性能高。
- 灵活的文档模型给开发带来的方便。
- 水平扩展机制能轻松应对从百万到十亿级别的数据量处理，这也是 MongoDB 名字来源于单词 humongous（极大的）的原因。

笔者在下面的章节中会为读者详细介绍 MongoDB 的这些特点。

8.2 核心机制解析

本节介绍 MongoDB 高性能背后的两个机制：MMAP（内存文件映射）和 Journal 日志。

8.2.1 MMAP（内存文件映射）

MongoDB 使用了操作系统提供的 MMAP（内存文件映射）机制进行数据文件的读写，MMAP 把文件直接映射到进程的内存空间，这样文件就会在内存中有对应的地址，这时对文件的读写是能通过操作内存进行的，而不需要使用传统的如 fread、fwrite 文件操作方式。

传统的文件操作流程：某个进程要读取硬盘上的数据，需要先把硬盘上的数据复制到内核缓冲区，再复制到进程的内存空间，如图 8-1 所示。

通过 MMAP，可以把文件直接映射到进程的内存空间中，如图 8-2 所示。

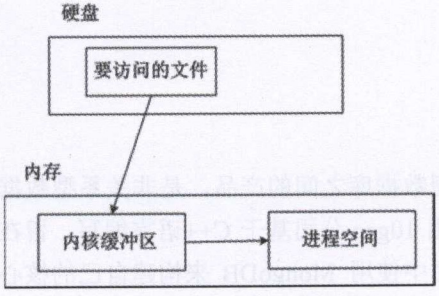


图 8-1 传统的文件读取流程

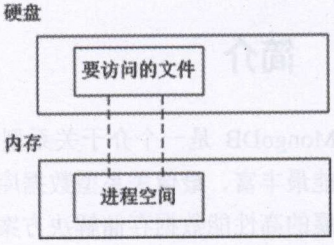


图 8-2 MMAP 文件读取流程

读者需要注意：MMAP 只是把文件映射到进程空间，并不是全部映射到内存，只有访问到的数据才会被操作系统转移到内存。但是因为内存是有限的，MongoDB 虽然可以存储比内存更大的数据，但是对于热数据（也就是需要存放在内存中的数据）不建议超过内存的大小。当热数据超过内存的大小，操作系统内存管理机制会把使用频率最低的数据换到交换分区

（Swap），这种来回的数据切换严重影响了 MongoDB 的性能。

当 MongoDB 需要插入新记录时，通过 MMAP 把数据库文件映射到内存后进行操作。MongoDB 默认是每分钟把内存中映射的数据刷到磁盘，也可以通过启动参数“--syncdelay”控制这个频率。

另外，即使 MongoDB 不把内存映射的数据刷到磁盘，操作系统也会定期把修改过的数据刷到磁盘，Linux 的 dirty_writeback_centisecs 参数用于定义脏数据在内存停留的时间（默认为 500，即 5 秒），过了这个时间脏数据就会被系统刷到磁盘上。

MongoDB 使用了 MMAP 的机制把大量的文件操作都交给操作系统完成，大大减轻了 MongoDB 开发者的负担。

8.2.2　Journal 日志

读者通过阅读 8.2.1 节可以了解到 MongoDB 通过 MMAP 机制，数据会在内存中停留一段时间，如果在这段时间内系统宕机数据还没来得及刷到硬盘，那么这些修改的数据不就丢失了吗？

MongoDB 内部通过 Journal 日志解决了这个问题。MongoDB 的所有数据更新操作会记录并保存到 Journal 日志，Journal 日志保存在 dbpath 路径中的 Journal 文件夹中。当系统宕机后 MongoDB 重启时，通过 Journal 日志上的操作记录，就能把数据恢复。

8.3　入门

MongoDB 的设计思想和 MySQL 是有很大区别的，其基于 MongoDB 的文档型系统，衍生出一系列不同的特性，由此也产生了有别于 MySQL 的应用场景。

MongoDB 是面向文档的数据，放弃关系模型的原因是为了更灵活的扩展性，当然还有别的好处，例如，MySQL 中为了方便增加某个产品的额外属性，需要把产品的额外属性拆分到另外一张表中，以便进行连接查询。因此，对于额外属性的增加、删除、修改都很麻烦，需要更多额外的操作。

移动互联网项目需求经常变动和发展，关系模型的僵硬性有时不太适合这种项目。更何况移动互联网项目经常需要在线修改数据表的结构，对于上千万甚至上亿规模的数据表来说，里面的风险和对在线服务的影响很大。

用 MongoDB 就能很简单地解决上面的问题。

MongoDB 把关系模型转变为文档模型，基本思路是把原来的行变为更灵活的文档模型，文档的键不是固定的，也不会事先定义。文档是 MongoDB 的核心，多个键值组合在一起就是文档。文档以 bjosn 格式存储，bson 基于 JSON 格式，MongoDB 选择 JSON 进行改造的主要原因是 JSON 的通用性及 JSON 的无模式的特性。

MongoDB 中集合就是一组文档，如果说 MongoDB 中的文档类似于关系数据的行，那么集合就类似于表。

MongoDB 中多个文档组成集合，同样多个集合也会组成数据库。一个 MongoDB 实例中可以有多个数据库，它们是相互独立的，每个数据库有独立的权限设置。每个数据库名最终会变成硬盘上的文件名。MongoDB 中下面的 3 个数据库名是保留的：admin、local、config。

最后总结一下 MongoDB 的数据结构与 MySQL 的数据结构的映射对应关系。

MongoDB	MySQL
文档	数据行
集合	数据表
数据库	数据库

8.3.1 基本操作

为了让没接触过 MongoDB 的读者对 MongoDB 有个初步的认识，笔者在本节中将会介绍 MongoDB 的基本操作。

MongoDB 有两个基本的组件。

- mongod：MongoDB 服务端程序，启动 MongoDB 的服务。
- mongo：MongoDB 客户端程序，连接 mongod 服务端进行相关的管理工作。

当 MongoDB 服务端启动后，通过 mongo 客户端程序连接 MongoDB 服务端，默认是连接本地 MongoDB 服务器的默认端口。

```
[root@jeff ~]# /usr/local/mongodb/bin/mongo
MongoDB shell version: 3.0.5
connecting to: test
```

客户端连接后会选择默认数据库“test”，用户切换到别的数据库，可以使用命令：

```
use 数据库名
```

1. 插入文档

下面演示怎么在集合中插入两个文档，要注意文档是一个 JSON 的扩展（bson）式。

```
>db.person.insert({"name":"jeff","age":25})
>db.person.insert({"name":"tom","age":26})
```

上面两条命令，分别在数据库 test 的集合 person 下插入了两个文档信息，读者可能已经注意到，集合 person 也是不需要手动创建的，当 MongoDB 使用到这个集合时发现不存在就会自动创建这个集合。

2. 查找文档

插入数据后为了让读者更早体验 MongoDB 数据存储的特点，在下面的示例中把刚才插入的两个文档查出来。

```
>db.person.find()
{ "_id" : ObjectId("55cd29d229866714a4b512ab"), "name" : "jeff", "age" : 25 }
{ "_id" : ObjectId("55cd2b1729866714a4b512ad"), "name" : "tom", "age" : 26 }
```

读者注意到 MongoDB 为每个文档生成了“_id”，这个是 GUID（全局唯一标识）值，保证了文档的唯一性。

MongoDB 也根据某个 Key 的值查找文档，例如，查找“name”为“jeff”的文档可以用如下的命令。

```
>db.person.find({"name":"jeff"})
{ "_id" : ObjectId("55cd29d229866714a4b512ab"), "name" : "jeff", "age" : 25 }
```

MongoDB 的查找命令也提供了类似于 MySQL 的<、<=、>、>=、!=等操作，在 MongoDB 中分别对应$lt、$lte、$gt、$gte、$ne。

下面的示例返回符合条件“age >25” 的文档。

```
>db.person.find({"age":{"$gt":25}})
{ "_id" : ObjectId("55cd2b1729866714a4b512ad"), "name" : "tom", "age" : 26 }
```

$in 操作符等同于 SQL 中的 in，下面的示例等同于 SQL 中的 in (25)。

```
>db.person.find({"age":{"$in":[25]}})
{ "_id" : ObjectId("55cd29d229866714a4b512ab"), "name" : "jeff", "age" : 25 }
```

$or 操作符等同于 MySQL 的 or，$or 的条件放在一个数组中，每个数组元素表示 or 的一个条件，下面的示例等同于 name=“jeff” or age=26。

```
>db.person.find({"$or":[{"name":"jeff"},{"age":26}]})
{ "_id" : ObjectId("55cd29d229866714a4b512ab"), "name" : "jeff", "age" : 25 }
{ "_id" : ObjectId("55cd2b1729866714a4b512ad"), "name" : "tom", "age" : 26 }
```

3. 更新文档

更新文档的方法的第一个参数是查找的条件，第二个参数是更新的值。

例如，把“name”为“jeff”的文档的“age”更新为26，可用如下的命令：

```
>db.person.find({"name":"jeff"})
{ "_id" : ObjectId("55cd29d229866714a4b512ab"), "name" : "jeff", "age" : 25 }
>db.person.update({"name":"jeff"},{"name":"jeff","age":26})
>db.person.find({"name":"jeff"})
{ "_id" : ObjectId("55cd29d229866714a4b512ab"), "name" : "jeff", "age" : 26 }
```

4. 删除文档

删除文档的方法如果带参数，就是删除符合特定条件的文档，如果不带参数，就是把集合中所有的文档删除，读者使用该命令的时候需要特别谨慎。

下面的示例演示了带参数的删除方法和不带参数的删除方法：

```
>db.person.remove({"age":26})
WriteResult({ "nRemoved" : 1 })
>db.person.find()
{ "_id" : ObjectId("55cd29d229866714a4b512ab"), "name" : "jeff", "age" : 25 }
>db.person.remove({})
>db.person.find()
```

8.3.2 数组操作

MongoDB 也支持数组的操作，在下面的示例中，插入 3 个数组。

```
>db.arr.insert({ "fruit" : [ "Apple", "banana", "peach" ] })
>db.arr.insert({ "fruit" : [ "Apple", "banana", "orange" ] })
>db.arr.insert({ "fruit" : [ "Apple", "cherry", "orange" ] })
```

下面的示例，演示查找包含“banana”的数组。

```
>db.arr.find({"fruit":"banana"})
{ "_id" : ObjectId("55ce88dd678cb85541f1a290"), "fruit" : [ "Apple",
"banana", "peach" ] }
{ "_id" : ObjectId("55ce88ea678cb85541f1a291"), "fruit" : [ "Apple",
"banana", "orange" ] }
```

检索数组中需要包含多个元素的情况，需要使用$all。在下面的示例中，数组必须同时包含“Apple”和“orange”。

```
>db.arr.find({"fruit":{"$all":["Apple","orange"]}})
{ "_id" : ObjectId("55ce88ea678cb85541f1a291"), "fruit" : [ "Apple",
"banana", "orange" ] }
{ "_id" : ObjectId("55ce88f9678cb85541f1a292"), "fruit" : [ "Apple",
"cherry", "orange" ] }
```

数组中的元素也可以内嵌，例如，一个商品的信息文档中包含商品的名称、价格，还有购

买者对这个商品的评分（1~5 分），MongoDB 的示例如下。

```
   >db.product.insert({"name":"shirt","price":200,"comments":[{"author":"to
m","score":3},{"author":"jeff","score":5}]})
   >db.product.insert({"name":"suit","price":120,"comments":[{"author":"ter
ry","score":4},{"author":"jeff ","score":2}]})
   >db.product.insert({"name":"coat","price":100,"comments":[{"author":"jon
y","score":3},{"author":"jeff ","score":4}]})
   >db.product.find()
   { "_id" : ObjectId("55ce903cae319b66d78a494a"), "name" : "shirt",
"price" : 200, "comments" : [ { "author" : "tom", "score" : 3 }, { "author":
"jeff", "score" : 5 } ] }
   { "_id" : ObjectId("55ce9043ae319b66d78a494b"), "name" : "suit",
"price" : 120, "comments" : [ { "author" : "terry", "score" : 4 },
{ "author" : "jeff ", "score" : 2 } ] }
   { "_id" : ObjectId("55ce904cae319b66d78a494c"), "name" : "coat",
"price" : 100, "comments" : [ { "author" : "jony", "score" : 3 },
{ "author" : "jeff ", "score" : 4 } ] }
```

当检索集合 product 时，需要以“comments”中的元素为检索条件，可以使用$elemMatch 操作符。在下面的示例中，检索购买者为“jeff”且评分大于 4 的商品。

```
   >db.product.find({"comments":{"$elemMatch":{"author":"jeff","score":{"$g
t":4}}}})
   { "_id" : ObjectId("55ce903cae319b66d78a494a"), "name" : "shirt",
"price" : 200, "comments" : [ { "author" : "tom", "score" : 3 }, { "author":
"jeff", "score" : 5 } ] }
```

8.3.3　实例演示 MySQL 和 MongoDB 设计数据库的区别

下面列举了一个数据库设计的例子，对比了 MySQL 和 MongoDB 两种数据库结构设计上的不同。

本节中所举的例子，是一个简化版的设计，希望借此给读者们扩展一下思路，介绍 MySQL 和 MongoDB 在设计思维上不一样的地方。

假设要维护一个商品信息库，里面除了包含商品名称和价格这两个基本属性外，对于不同的商品，还要包含其他额外的属性，例如，如果商品是衣服，要包含颜色、尺寸这两个属性，如果商品是手机，还要包含后置摄像头像素、机身内存这两个属性。

1. 使用 MySQL 的例子

手机的基本属性表是“product”，其他属性由于差异比较大，需要采用一个额外参数表“product_params”的形式来设计，数据表的设计如下所示：

```
/* 商品基本属性表 */
CREATE TABLE IF NOT EXISTS `product` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `name` VARCHAR(100) NOT NULL,
    `price` FLOAT(10,2) NOT NULL,
    PRIMARY KEY (`id`)
);
 /* 商品额外属性表 */
CREATE TABLE IF NOT EXISTS `product_params` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `product_id` int(10) unsigned NOT NULL,
    `name` varchar(100) NOT NULL,
    `value` varchar(100) NOT NULL,
    PRIMARY KEY (`id`)
);
```

下面插入商品信息。

```
INSERT INTO `product` (`id`, `name`, `price`) VALUES
(1, '大衣', 110.00),
(2, 'iPhone 6',4788.00 ),
(3, '小米 红米 2A',499.00 ),
(4, '小米 Note 全网通',2099.00 ),
(5, '酷派 大神 F2',699.00 ),
(6, '华为 P8 青春版',1588.00 );
```

插入商品的附加信息。

```
INSERT INTO `product_params` (`id`, `product_id`, `name`, `value`)
VALUES
(1, 1, '颜色', '红'),          /* 大衣的颜色 */
(2, 1, '尺寸', 'S'),           /* 大衣的尺寸 */
(3, 2, '机型', 'iPhone'),      /* iPhone 6 的机型 */
(4, 2, '机身内存', '16'),      /* iPhone 6 的机身内存 */
(5, 3, '机型', 'Android'),     /* 小米 红米 2A 的机型 */
(6, 3, '机身内存', '8'),       /* 小米 红米 2A 的机身内存 */
(7, 4, '机型', 'Android'),     /* 小米 Note 全网通的机型 */
(8, 4, '机身内存', '16'),      /* 小米 Note 全网通的机身内存 */
(9, 5, '机型', 'Android'),     /* 酷派 大神 F2 的机型 */
(10, 5, '机身内存', '16'),     /* 酷派 大神 F2 的机身内存 */
(11, 6, '机型', 'Android'),    /* 华为 P8 青春版的机型 */
(12, 6, '机身内存', '16');     /* 华为 P8 青春版的机身内存 */
```

注意：为了简化这个例子，上面的 MySQL 表没有严格遵从关系型数据库的范式设计，把字符和数值统一保存为字符串类型，在实际查找过程中，MySQL 允许在字符串上进行数值类型的查询，只是会影响性能。

如果需要查找机型为 Android、机身内存为 16GB、价格大于 1000 元的手机，需要按照下面的方式查找。

首先，在商品额外属性表“product_params”中查找机型为“Android”、机身内存等于 16GB 的商品的 id。

```
SELECT product_id FROM `product_params` WHERE name = '机型' AND value =
'Android';
SELECT product_id FROM `product_params` WHERE name = '机身内存' AND value
= 16;
```

使用上面两个查询结果的交集可以获得机型为“Android”、机身内存等于 16GB 的商品的 id 为[4,5,6]，然后在“商品基本属性表”中查找出价格大于 1000 元的手机：

```
SELECT * FROM `product` WHERE price>1000 and id in(4,5,6)
```

得到的结果如图 8-3 所示。

id	name	price
4	小米 Note 全网通	2099.00
6	华为 P8青春版	1588.00

图 8-3　MySQL 查询结果图

2. 使用 MongoDB 的例子

用 MongoDB 处理商品信息的数据，得益于 MongoDB 无范式的特点，用文档就能很简单地存储衣服和手机这两种不同的商品信息，MongoDB 的命令如下。

```
db.product.insert({"name":"大衣",
                   "price":110.00,
                   "params":[
                   {"name":  "尺寸", "value": "S"},
                   {"name": "颜色", "value": "红"},
                   ]
                  })
db.product.insert({"name":"iPhone 6",
                    "price":4788.00,
                    "params": [
                    {"name": "机身内存", "value": 16},
                    {"name": "机型", "value": "iPhone"}
                    ]
                   })
```

```
db.product.insert({"name":"小米 红米 2A",
                   "price":499.00,
                   "params": [
                   {"name": "机身内存", "value": 8},
                   {"name": "机型", "value": "Android"}
                   ]
                  })
db.product.insert({"name":"小米 Note 全网通",
                  "price":2099.00,
                  "params": [
                  {"name": "机身内存", "value": 16},
                  {"name": "机型", "value": "Android"}
                  ]
                 })
db.product.insert({"name":"酷派 大神 F2",
                  "price":699.00,
                  "params": [
                  {"name": "机身内存", "value": 16},
                  {"name": "机型", "value": "Android"}
                  ]
                 })
db.product.insert({"name":"华为 P8 青春版",
                  "price":1588.00,
                  "params": [
                  {"name": "机身内存", "value": 16},
                  {"name": "机型", "value": "Android"}
                  ]
                 })
```

如果需要查找机型为“Android”、机身内存为 16GB、价格大于 1000 元的手机，则用前面介绍过的$gt（相当于 MySQL 的“>”），$all（相当于 MySQL 的“and”），$elemMatch 这些操作符就能完成查找。

```
db.product.find({"params": {
                 "$all":[
                     {$elemMatch: {"name": "机型", "value": "Android"}},
                     {$elemMatch: {"name": "机身内存", "value": 16}}
                 ]
             },
             "price":{"$gt":1000},
            })
```

查找的结果如下。

```
{ "_id" : ObjectId("55d1d874ece7a0dc06b2c172"), "name" : "小米 Note 全网通
```

```
", "price" : 2099, "params" : [ { "name" : "机型", "value" : "Android" },
{ "name" : "机身内存", "value" : 16 } ] }
    { "_id" : ObjectId("55d1d874ece7a0dc06b2c174"), "name" : "华为 P8 青春版",
"price" : 1588, "params" : [ { "name" : "机型", "value" : "Android" },
{ "name" : "机身内存", "value" : 16 } ] }
```

8.4　高可用集群

MongoDB 作为 NoSQL 的代表之一，其自身具备了良好的扩展性，能快速搭建一个高可用、可扩展的分布式集群，下面介绍搭建 MongoDB 集群的几种方案。

8.4.1　主从

MongoDB 采用双机主从备份，主节点的数据会自动同步到从节点，当主节点宕机后，能切换到从节点继续提供服务，如图 8-4 所示。

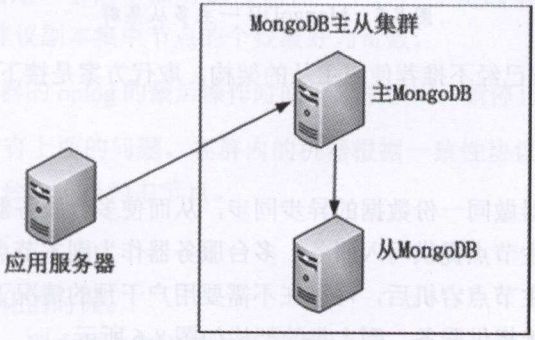


图 8-4　MongoDB 主从集群

当服务器访问量上升后，可能单靠一台主节点提供服务会造成比较大的性能瓶颈。在大多数的业务中，数据读写的比例会达到 8:2，甚至是 9:1，访问的压力集中在读取数据方面，当一个节点无法承受读压力，可以把一个节点增加为多个节点来减轻单台服务器的负载。MongoDB 提供了一主多从的架构来降低单台节点的负载，由主节点负责写数据，多个从节点负责读数据，数据从主节点复制到多个从节点，如图 8-5 所示。

但是 MongoDB 的主从架构有下面 3 个问题。

- 当主节点宕机后，不能支持自动切换连接，目前只能手动切换。
- 主节点写压力过大的问题没法解决。
- 没法支持数据的路由。

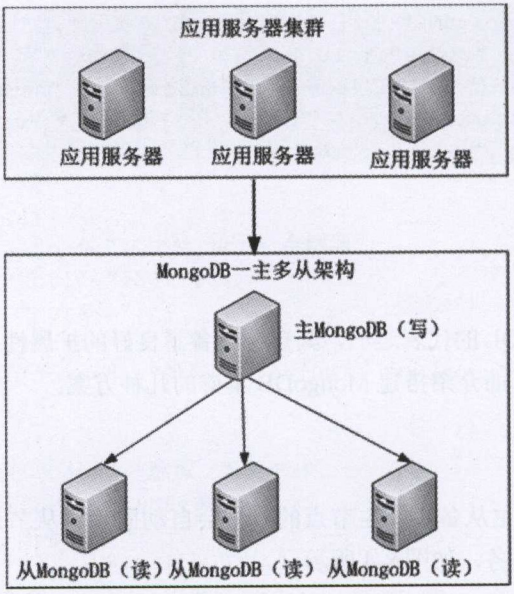


图 8-5　MongoDB 一主多从集群

现在 MongoDB 官方已经不推荐使用主从的架构，取代方案是接下来介绍的副本集。

8.4.2　副本集

副本集使用多台机器做同一份数据的异步同步，从而使多台服务器拥有同一份数据的多个副本。一台服务器作为主节点提供写入服务，多台服务器作为副本节点提供读取服务，实现读写分离和负载均衡。当主节点宕机后，可以在不需要用户干预的情况下把一台副本节点或其他节点提升为主节点，继续提供服务。副本集的架构如图 8-6 所示。

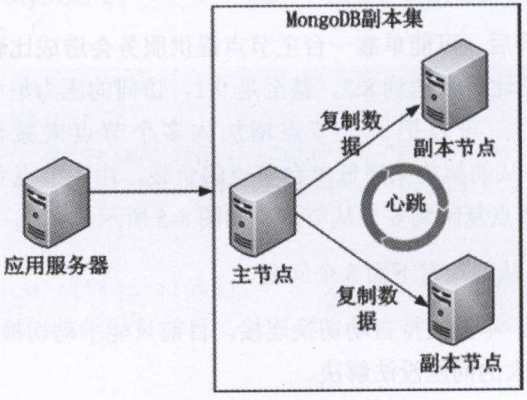


图 8-6　MongoDB 副本集架构图

应用服务器连接到整个服务集，并不关心主服务器是否挂掉。主节点负责整个副本集的读写，副本节点从主节点中同步数据。副本集内的机器通过心跳机制通信，当检测到主节点宕机时，副本集内的服务器从剩余的服务器中选举一台新的服务器作为主节点继续提供服务。这一切对于应用服务器来说都是透明的。

MongoDB 的副本集是通过 oplog 实现的，主节点数据的修改操作会被记录到主节点的 oplog 日志，然后副本节点通过异步方式复制主节点的 oplog 文件并且将 oplog 日志应用到副本节点，从而实现了与主节点的数据同步。

副本集启动时，副本集内的服务器通过选举机制选举一台服务器为主节点，其他服务器为副本节点，选举过程如下。

1. 获取每台服务器 oplog 的最后操作时间。在 MongoDB 中，修改的数据会先放在内存中一段时间再写入硬盘，为了防止未写入硬盘前因为断电等原因造成数据丢失，所以 MongoDB 会把相关的数据更新操作写入日志 oplog 中，以便 MongoDB 宕机后恢复。
2. 如果集群中有超过一半的节点宕机（包含一半），停止选举。为了避免这个问题，MongoDB 官方建议副本集中节点的个数最好为奇数。
3. 如果集群中服务器的 oplog 的最后操作时间看起来很旧，就停止选举等待管理员操作。
4. 如果集群内都没有上面的问题，集群内的机器根据一致性协议，选举 oplog 的最后操作时间最新的那台服务器为主节点。

选举触发的条件如下。

- 副本集刚刚初始化的时候。
- 由于网络的原因，副本节点和主节点断开通信。
- 主节点宕机。

副本集的集群中，有如下 4 种角色。

- Primary：主节点，负责集群的读写。
- Secondary：副本节点，从 Primary 的 oplog 读取操作日志，以便与 Primary 保持一致，主要是备份。
- Arbiter：仲裁节点，其不负责任何读写，只负责主节点宕机的时候的参与选举。
- Passive Node：除了没有被选举权，其他同 Secondary。

MongoDB 官方推荐的集群节点是奇数，同时集群中又提供了仲裁节点这个角色，因此为了保证副本集的选举能顺利进行，可以在集群中加入一台仲裁服务器，如图 8-7 所示。

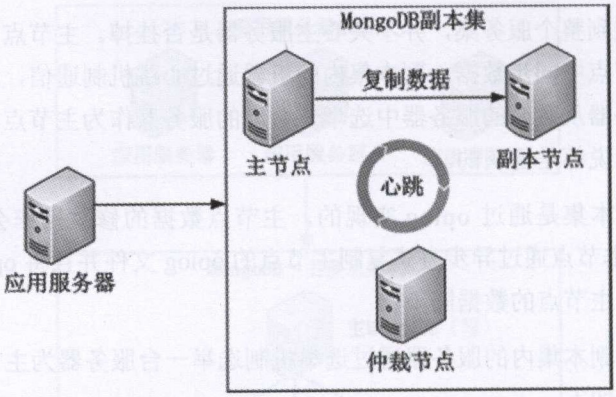


图 8-7　副本集中加入仲裁节点的架构图

副本集还有一个问题：如果主节点读写压力过大，为了减轻主节点的压力，可以设置读写分离，由主节点负责读写，副本节点负责读，仲裁节点只参与选举，如图 8-8 所示。

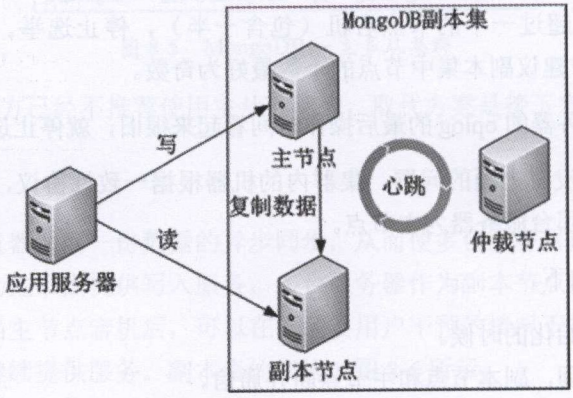


图 8-8　MongoDB 读写分离的架构

8.4.3　分片

随着 MongoDB 数据量和访问量持续增加，单个集群的性能有可能达到瓶颈，针对这种情况，架构上一般的处理方法是“分而治之”，把集群中大量的数据读写请求分散到多个集群处理，在 MySQL 中称为数据库分片。

MySQL 要实现分库功能，还要依赖 Amoeba、Cobar 或 MyCAT 等分布式关系数据库产品，而 MongoDB 提供了分片这种原生的机制来处理这种“分而治之”的问题。

当一台服务器的承载能力达到瓶颈，无论怎样提升单机硬件配置（例如加 CPU、内存，把硬盘换成 SSD）都无法解决问题，这时最好的策略就是分片，把集中在一台服务器上的压力

分散到多台。例如，有性能瓶颈的服务器要处理 2TB 的数据，而把 2TB 数据通过合理的分片策略分散到两台服务器上，每台服务器就存储 1TB 的数据和承担一半的访问量了。

为了保证每个分片服务器能均衡地承担访问量，避免有的服务器承担很大的访问量，有的服务器承担很少的访问量，需要在设置分片规则时就仔细考虑。例如，根据文档的 id 来分片，就能保证分片是比较均衡的。

MongoDB 分片的架构图如图 8-9 所示。

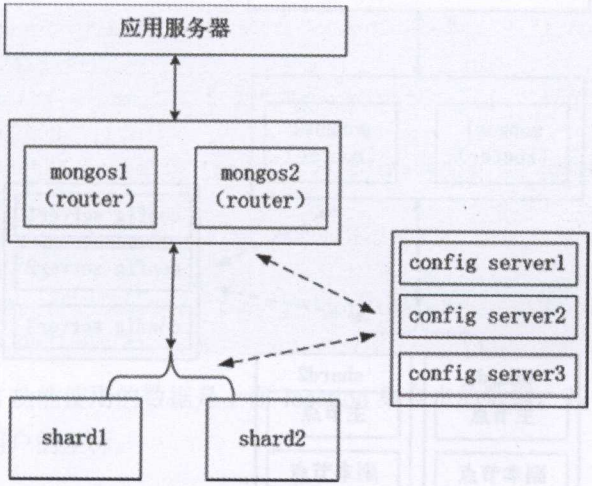


图 8-9　MongoDB 分片的架构图

在图 8-9 中，MongoDB 通过下面的 3 个组件实现分片。

- mongos：作为数据库集群请求的入口，由于数据已经分布在 shard 服务器上，所有请求经过 mongos 转发到 shard 服务器上，mongos 充当路由的角色。mongos 是无状态的，因此可以部署多台 mongos 做负载均衡，防止因为某台 mongos 宕机导致整个集群不可用。在某些 MongoDB 客户端中，连接 MongoDB 集群时支持同时传入多个 mongos ip 地址作为参数，在客户端内部实现负载均衡和故障移除。
- config server：配置服务器，存储了所有数据库元信息（路由、分片）的配置。mongos 服务器自身是没有在硬盘上存储 shard 服务器和路由的元信息的，只是在内存中存储过，当 mongos 服务器重启或关机后，这些信息会丢失。因此，为了保证 shard 服务器和路由的元信息不丢失，信息会存储在 config server 服务器。当 mongos 第一次启动或重启时，会从 config server 加载配置信息，同时，当配置信息发生变化时，mongos 服务器会接收到最新的变化并更新内存中的数据。由于配置服务器中存储的信息太重要，万一丢失会引起整个集群的崩溃，所以在生产环境中会配置多台 config server 保证高可用。

- shard server：分片服务器，分片后保存数据的服务器。

在分片集群中，某个分片的数据只存储在一个服务器，如果这个服务器宕机了，那这部分数据就无法访问。为了保证分片的高可用，分片服务器也能使用副本集的架构，在生产环境中，每个副本集通常是由一个主节点（Primary）、一个副本节点（Secondary）、一个仲裁节点组成（Arbiter），架构如图 8-10 所示。

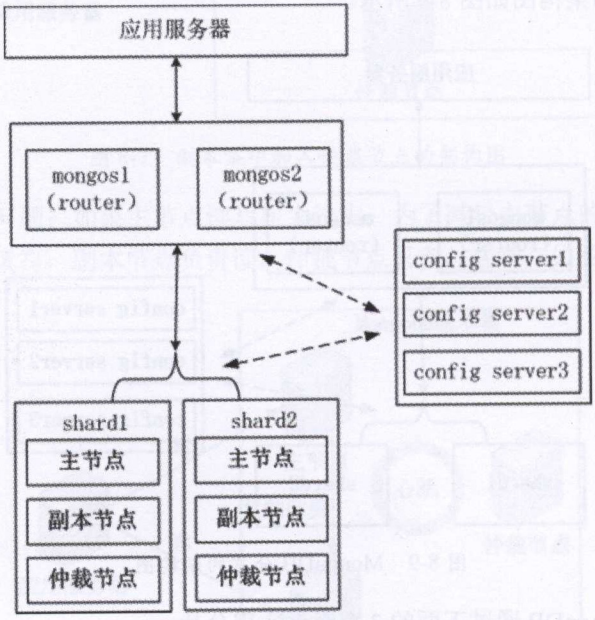


图 8-10　分片加上副本集的架构

在 UCloud 上已经提供了基于 MongoDB 的分片集群的云数据库服务，但是笔者写本节的时候（2015 年 8 月 19 日），UCloud 只提供了配置服务器和分片服务器组件，mongos 还需要用户在云服务器上自行搭建，希望 UCloud 尽快把服务做得更完善。

参考资料：

https:// docs.MongoDB.org/manual/core/sharding-introduction/

8.5　LBS——地理位置查询

基于 LBS 的 App，必不可少的一个功能是根据当前用户的位置查找附近范围内的人或物。这些 App 的数据中都包含了地理位置信息，而如何处理、分析、存储这些地理位置信息，则

是 LBS 技术的一个关键点。

关于 LBS 技术的详细分析，可查看本书“9.3 LBS App 后台架构”这一节。因为 MongoDB 封装了 LBS 常用的操作，全球流行的 LBS 应用 foursquare，国内的快的打车和滴滴打车（现在这两家公司已合并）都曾经选择了 MongoDB 处理 LBS。

在本节中将演示如何使用 MongoDB 处理 LBS。

首先在 MongoDB 中插入地理位置的数据示例如下。

```
   db.location.insert({ "name" : "jeff","coordinate" : { "longitude" :113.392237,
"latitude" : 23.062429 }})
   db.location.insert({ "name" : "tom","coordinate" : { "longitude" : 113.392354,
"latitude"  : 23.062582 }})
   db.location.insert({ "name" : "mike","coordinate" : { "longitude" :113.392107,
   "latitude" : 23.062246 }})
   db.location.insert({ "name" : "lili","coordinate" : { "longitude" : 113.39972,
"latitude" : 23.067753 }})
   db.location.insert({ "name" : "lucy","coordinate" : { "longitude" :113.399343,
   "latitude" : 23.067158 }})
```

本节演示的 LBS 功能使用的数据是上面 location 集合中的数据，其中 name 表示用户的名称，coordinate 表示用户的坐标。

MongoDB 原生支持地理位置索引，可以直接用于位置距离计算和查询，常用的地理位置索引有两种。

- 2d：平面坐标索引，适用于基于平面的距离计算。
- 2dsphere：几何球体索引，适用于球面上的距离计算。

在地理位置数据上建立地理位置索引：

```
   > db.location.ensureIndex({'coordinate':'2d'})
```

或者：

```
   > db.location.ensureIndex({'coordinate':'2dsphere'})
```

如果要追求最准确的精度，应该选择 2dsphere，在坐标跨度不大的情况下（例如几百、几千千米），这两者的计算结果相差无几。MongoDB 官方推荐使用 2dsphere。

下面演示了 LBS 应用中最常用的“查找附近的人（物）”的操作，同时也演示了如何查找一个特定区域内的坐标。

1. 使用 MongoDB 查找附近的人

在 LBS 中，应该会遇到如图 8-11 所示的需求，把当前用户附近的人（物）在地图中显示出来。

图 8-11　显示当前用户附近的人（物）

查找附近的人（物）的功能，可以等同于在 MongoDB 中，查找有哪些坐标和用户当前坐标直线距离为某个固定值以内（例如 1000 米范围内）。MongoDB 使用“$nearSphere”操作符，一条查询语句就能把这个功能实现。

下面的语句表示查找集合 location 中，和坐标[113.392237, 23.062429]的间距在 1 千米范围内的坐标：

```
db.location.find({"coordinate": {
        "$nearSphere": [113.392237, 23.062429],
        "spherical": true,
        "$maxDistance": 1000/6378137,
        "distanceMultiplier": 637813
    }
})
```

上面“spherical”为 true 表示计算距离，distanceMultiplier 指定地球半径来得到实际的千米或者米的距离，“1000/6378137”就是把距离转化为弧度。

查找 500 米范围，可使用如下的命令：

```
db.location.find({"coordinate": {
```

```
            "$nearSphere": [113.392237,23.062429],
            "spherical": true,
            "$maxDistance": 500/6378137,
            "distanceMultiplier": 637813
        }
    })
```

还有一种很常见的需求，把查找到的坐标按照距离排序，并标明和当前用户坐标的距离，如图 8-12 所示。

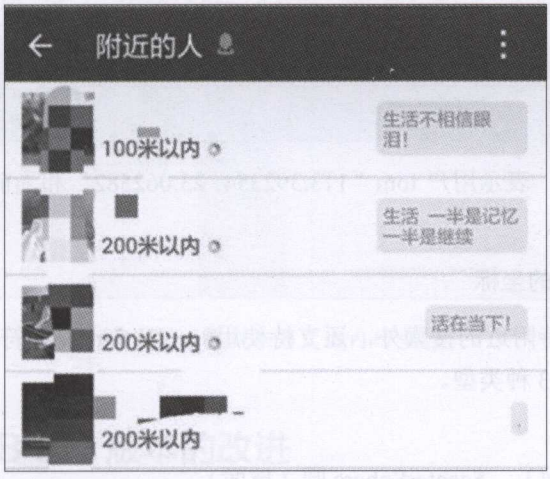


图 8-12　标注距离的附近的人

这个需求和前面的查找附近的人很相近，但是返回结果需要包含距离，MongoDB 提供了 geoNearmm 命令来解决这个需求。geoNear 的返回结果中包含了距离的信息。

geoNear 命令是基于 db 的 command，而不是基于 collection 的 find，也就是需要通过 runcommand 执行。在查找集合 location 中，和坐标[113.392237, 23.062429]的间距在 500 米范围内的坐标，同时显示和目标坐标的距离，整个命令如下：

```
db.runCommand({
    "geoNear": "location",
    "near": [113.392237,23.062429],
    "spherical": true,
    "$maxDistance": 500/6378137,
    "distanceMultiplier": 6378137
})
```

在上面的命令中，geoNear 指定了要操作的集合，之所以在命令中没出现 coordinate 这个存储了坐标的 Key，是因为在一个集合中只能在一个 Key 上建立地理坐标索引，所以

MongoDB 会自动查找到这个索引。

返回的结果是有序的，包含了类似下面的信息：

```
{
            "dis" : 20.825177011762033,
            "obj" : {
                "_id" : ObjectId("55d6da748271c3157a70988d"),
                "name" : "tom",
                "coordinate" : {
                    "longitude" : 113.392354,
                    "latitude" : 23.062582
                }
            }
        },
```

返回值中 dis 为 20，表示用户 tom“113.392354, 23.062582”和当前用户 jeff“113.392237, 23.062429”距离 20 米。

2. 查找某个范围内的坐标

MongoDB 除了支持附近的搜索外，还支持使用$geoWithin 操作符搜索某个范围内的坐标，这个操作符支持下面的 3 种类型。

- $box 矩形
- $center 圆（平面）、$centerSphere 圆（球面）
- $polygon 多边形

其中搜索$center 圆（平面）、$centerSphere 圆（球面）区域内的坐标，和查找某个用户附近的人（物）的坐标是相似的，都是指定圆心的坐标，通过指定半径得到圆的范围。区别是前者返回的结果是无序的，后者返回的结果是有序的。

下面以$box 矩形为例说明，假设如图 8-13 所示，要在 MongoDB 集合中查找黑色矩形区域内的坐标。

定义一个矩形范围的范围，根据图 8-13 指定矩形一个对角的坐标“113.39168，23.062726”、“113.392848，23.061973”，使用如下的查询语句就能查找 location 集合中矩形区域内的坐标：

```
db.location.find({
    "coordinate": {
        "$geoWithin": {
            $box :[ [113.39168,23.062726] , [113.392848,23.061973] ]
        }
    }
})
```

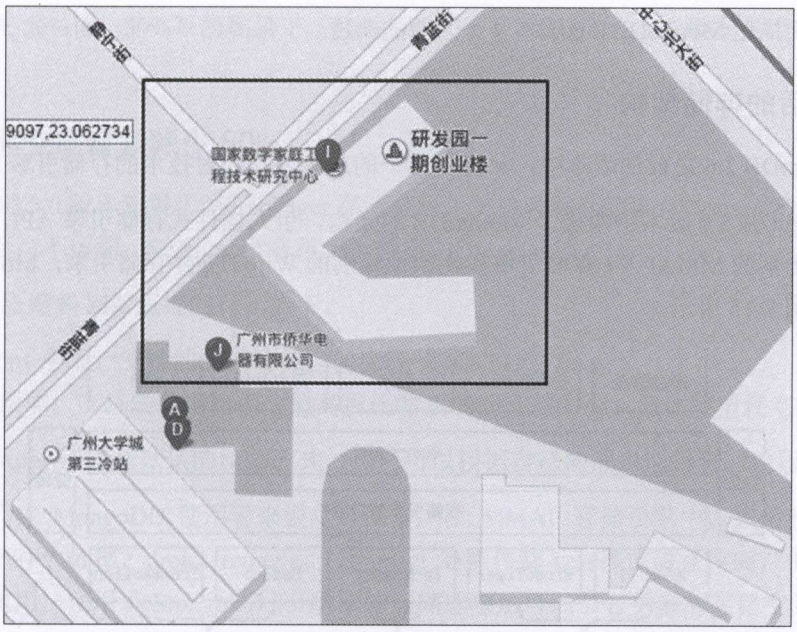


图 8-13　查找矩形区域内的坐标

8.6　MongoDB 3.0 版本的改进

MongoDB 3.0 版本于 2015 年 3 月 3 日正式发布，该版本标志着 MongoDB 的发展进入了一个全新的阶段。

上一个 MongoDB 版本号是 2.6，按照习惯这个版本号应该为 2.8，怎么一下子就跳到 3.0 了呢？据 MongoDB 大中华区首席技术顾问唐建法介绍，因为这个版本有了极大的改进，用 2.8 这个版本号有点委屈，因此 MongoDB 市场部主张命名为 3.0。

MongoDB 3.0 版本有 4 个方面的改进，如图 8-14 所示。

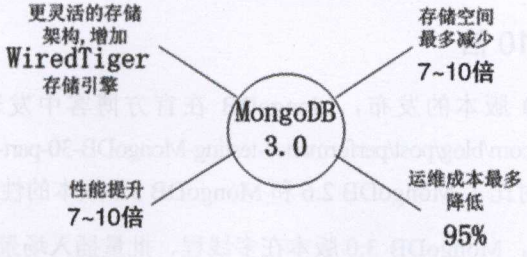


图 8-14　MongoDB 3.0 的新特性

下面分别描述 MongoDB 3.0 版本 4 个方面的改进。

8.6.1 灵活的存储架构

在 MongoDB 2.6 之前的版本中，只支持单一的基于内存映射技术的存储引擎 MMAP。

在 MongoDB 3.0 版本中改进了 MongoDB 的架构，引入插件式存储引擎 API，目前支持改进的集合锁级别的 MMAP V1 存储引擎和文档锁级别的 WiredTiger 存储引擎，MongoDB 3.0 版本的架构如图 8-15 所示。

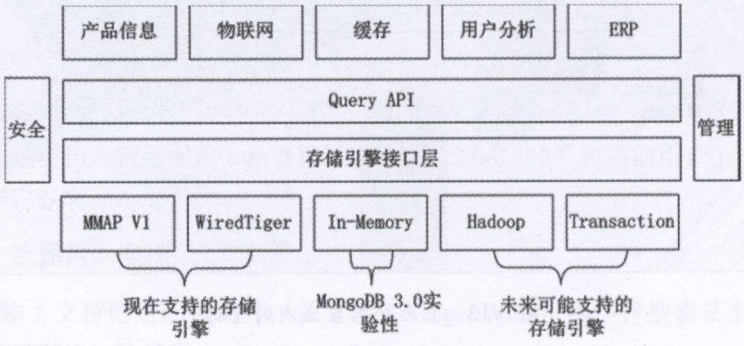


图 8-15 MongoDB 3.0 架构图

MongoDB 2.6 版本的 MMAP 存储引擎最大的问题是库锁，这意味着当多个客户端并发访问一个库时，如果某个客户端正在进行写操作，其他客户端都必须排队等待。MongoDB 达到了一定的并发量后，库锁对性能影响十分巨大，快的打车 App 曾经使用 MongoDB 处理 LBS，但因为 MongoDB 库锁问题，最后转为自研发 LBS 方案。

MongoDB 3.0 版本中 MMAP 存储引擎得到了改进，库锁变为集合锁。MongoDB 3.0 版本最核心的变化是增加了（收购而来）高性能、可伸缩的 WiredTiger 存储引擎，使 MongoDB 的性能得到前所未有的提升。特别是 WiredTiger 存储引擎实现了文档级别的锁，这意味着如果多个客户端同时更新一个集合内的多个文档，再也不需要因为库锁而排队等待了。

8.6.2 性能提升 7~10 倍

随着 MongoDB 3.0 版本的发布，MongoDB 在官方博客中发表了一篇性能测试报告（ https://www.MongoDB.com/blog/post/performance-testing-MongoDB-30-part-1-throughput-improvements-measured-ycsb），详细对比了 MongoDB 2.6 和 MongoDB 3.0 版本的性能差别。

在 YCSB 的测试中，MongoDB 3.0 版本在多线程、批量插入场景下，比 MongoDB 2.6 版本大约有 7 倍的增长。在 95%读、5%写的场景下，MongoDB 3.0 比 MongoDB 2.6 版本多 4 倍

的并发量。在 50%读、50%写的场景下，MongoDB 3.0 比 MongoDB 2.6 版本多大约 6 倍的并发量。

8.6.3　存储空间最多减少 80%

MongoDB 3.0 版本新增了 WiredTiger 存储引擎，支持对所有的集合数据进行压缩，支持的压缩选项包括：不压缩、SnAppy 压缩和 Zlib 压缩。

SnAppy 压缩和 Zlib 压缩的对比如下。

- SnAppy 压缩：支持的压缩比低，但比较节省 CPU 资源。
- Zlib 压缩：支持的压缩比高，最高可压缩 80%的文件大小，但比较消耗 CPU 资源。

其中 SnAppy 压缩是默认的压缩方式，用户可以根据自己的需求选择合适的压缩选项。

这对于广大 MongoDB 使用者来说是个福音，在 MMAP 存储引擎中，采用预分配机制分配大文件用于存放数据，drop、remove、 compact 等操作都不会释放磁盘空间，但是可以使预分配的空间重用。使用了 MongoDB 3.0 版本的压缩选项后可以节省更多的存储空间。

8.6.4　运维成本最多降低 95%

MongoDB 2.6 版本中如果用户使用了副本集、分片等集群方式，监控、备份和管理整个集群需要使用烦琐的命令。

MongoDB 3.0 版本的高级企业版中（MongoDB 3.0 包括社区版和高级企业版），新增了“Ops Manager”管理工具，通过里面集成的 RESTful API 和管理后台，让用户很方便地监控、备份和管理整个集群。

第 9 章
App 后台架构剖析

本章总结了四类 App 后台架构。

- 聊天 App 后台架构
- 社交 App 后台架构
- LBS App 后台架构
- 推送服务器后台架构

9.1 聊天 App 后台架构

聊天是目前移动互联网流行的通信方式，聊天功能是当今 App 后台功能中的重要部分。App 的聊天界面功能如图 9-1 所示。

图 9-1　App 的聊天界面

如图 9-1 所示，当用户 a 向用户 b 发送了一段消息后，用户 b 在线的话就能立刻收到消息。

研究聊天功能，就要先明白移动互联网有别于 PC 互联网的特性，从而更好地明白协议和架构背后的原理。

9.1.1　移动互联网的网络特性

当数据从网络的一端发送到另外一端（例如从 App 端到 App 后台），只有通过一个共同的约定（即协议），双方才能正确解析这些数据。协议就相当于一套语言（例如中文和英语），通信的双方必须知道每个数据是什么含义，才能准确解析数据。

因为移动互联网的特性，所以传统的 PC 通信协议不适用于移动互联网。下面从移动互联网的特性出发，详细讲述聊天的协议。

移动互联网有两个显著的特点。

- 弱网络性
- 对流量敏感

1. 弱网络性

弱网络性是指由于手机不断移动（例如在汽车、地铁上）的特性，有可能处于快速移动当中，因此出现信号不稳定、响应时间变长、出现经常丢包等情况。例如有时在地铁上，从 App 发送请求到 App 后台，App 后台处理完毕后返回数据这个过程因为延迟就可能需要 10 秒，该流程如图 9-2 所示。

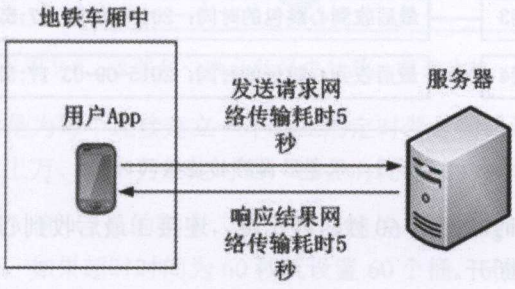


图 9-2　地铁车厢中 App 请求耗时

因此针对弱网络环境，开发者在设计协议时必须考虑尽量减少数据往返的次数，数据往返的次数越多，耗时越长。

另外由于弱网络性，App 以长连接的形式连接到服务器，可能会出现 App 和服务器的连接忽然中断的情况，而且这种情况是没法通过连接端口的异常判断。例如当地铁高速行驶导致

网络中断，App 没法向服务器的端口发送断开的信息，服务器还是以为和 App 一直保持着连接，这种现象称为 TCP half-open。

TCP half-open 的危害如下。

- 占用了服务器的资源（服务器维持一个连接是需要耗费内存的）。
- 发送消息的异常。

有效防止 TCP half-open 的方法是使用应用层心跳机制：在 App 和服务器保持连接的过程中，App 在规定时间间隔内向服务器发送一个数据（为了节省流量，数据可以只是一个字符“h”，因为是隔一定时间就发送一次，这种数据被形象地称为“心跳数据”）。服务器收到这个数据知道这个连接是有效的。对于那些超过规定时间间隔还未收到心跳数据的连接，服务器就主动断开连接，通过这种机制就能有效解决 TCP half-open 的现象。

服务器检查 App 的连接有 3 种方式。

（1）服务器记录每个连接收到心跳包的最后时间，每隔一段时间检查所有连接，如果发现某个连接最后收到心跳包的时间减去当前时间的值大于超时时间，就把那个连接断开。如图 9-3 所示，服务器的当前时间为“2015-09-03 17:53:02”，各个连接最后收到心跳包的时间如下。

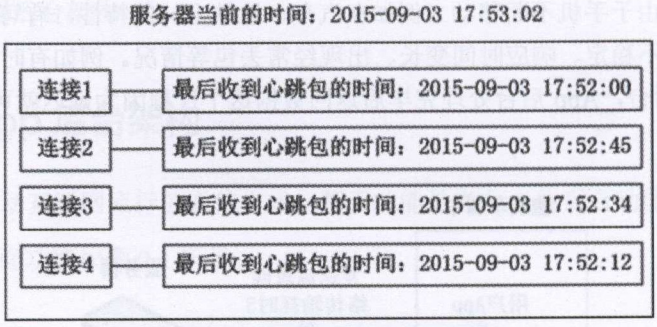


图 9-3　服务器保存的连接的时间

假设服务器设定的超时时间是 60 秒，经比较，连接 1 最后收到心跳包的时间超过了 60 秒，那么连接 1 将会被服务器断开。

这种方式的主要缺点是服务器需要同时检查所有连接，如果连接的数目很大（例如上万，上十万），检查连接时对系统的性能影响大。

（2）服务器为每个连接建立一个定时器，到了规定的超时时间没有收到心跳包定时器就触发把当前的连接断开，如果收到心跳包，就重设定时器，不断重复前面的过程。下面举一个例子说明。

当前服务器的时间是 2015-09-03 20:02:02，设定超时时间是 60 秒，当 App 和服务器建立连接，根据超时时间为这个连接设置了定时器的触发时间为 2015-09-03 20:03:02，如图 9-4 所示。

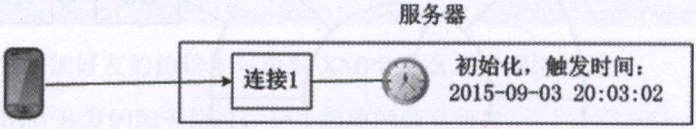


图 9-4　建立连接，初始化定时器

在 2015-09-03 20:03:00， App 向服务器发送了一个心跳包，服务器收到心跳包后把定时器的触发重设为 2015-09-03 20:04:00，如图 9-5 所示。

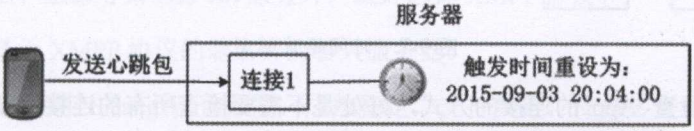


图 9-5　服务器收到心跳包后重设定时器

如果服务器到了定时的触发时间 2015-09-03 20:04:00 还没有收到 App 新发送的心跳包，则定时器被触发，主动断开了服务器和 App 的连接，如图 9-6 所示。

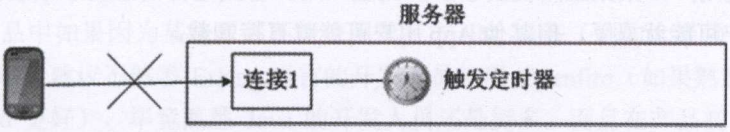


图 9-6　没收到心跳包触发定时器，断开连接

这种方式的主要缺点是为每个连接建立一个独立的定时器，如果连接的数目很大（例如上万、上十万），就会建立上万、上十万的定时器，也会消耗大量的系统资源。

（3）这种方式是（1）和（2）的折中，称为时间轮片（Timing Wheel）。按照超时时间的长短，每秒设置一个桶，如果超时时间为 60 秒就设置 60 个桶，60 个桶组成一个循环队列。第一个桶放一秒后将要超时的连接，第二个桶放两秒后将要超时的连接，每个连接一收到心跳包就把自己放在第 60 个桶，然后在每秒的定时器中把第一个桶的所有连接断开，把这个空桶挪到队尾。时间轮片的模型如图 9-7 所示。

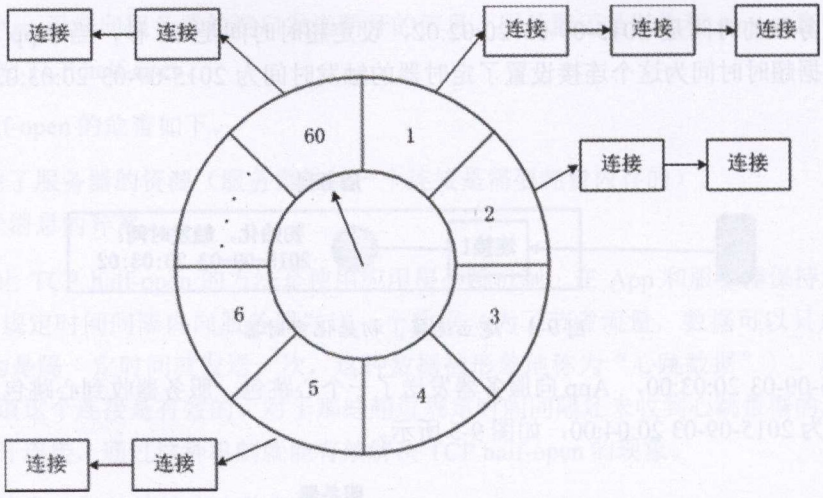


图 9-7　时间轮片模型

这种服务器检查 App 的连接的方式，好处是不需要检查所有的连接，节省了大量的系统资源。

2. 对流量敏感

在非 Wi-Fi 环境下用户使用手机卡的流量套餐上网，用户对流量比较敏感，各种安全监控 App 也经常提醒用户，某某应用耗费了多少流量，用户看到这种消息就非常紧张，如果是生活必备的 App 用户可能就忍了，但其他 App 用户可能就直接卸载。

9.1.2　协议

基于以上介绍移动互联网的特性，一个优秀的 App 通信协议有如下的要求。

- 简化系统设计。
- 提供可靠高效的消息传输，这个是重点关注的要求。
- 易于扩展需求。

现在常用的聊天协议如下。

- XMPP：一个基于 XML 的消息协议，其曾经被广泛应用于 Gtalk、Facebook 的聊天服务。

XMPP 协议作为一个被广泛使用的消息协议，有大量的网络资料和成熟开源模块，在 Android 和 iOS 平台上很方便就能集成 XMPP 协议。IM 作为一个复杂的系统有方方面面需要考虑，使用成熟的协议能帮助开发者避免很多问题，提高开发效率。

同时基于 XML 的 XMPP 协议的缺点也很明显：耗费流量。下面是 XMPP 协议的一部分。

```
<iq id="rosterset1" type="set">
<query XMLns="jabber:iq:roster">
<item jid="user@jabbercn.org" name="user"/>
</query>
</iq>
<presence from="contact@rooyee.biz" to=user@jabbercn.org type="subscribe"/>
```

从上面 XMPP 添加好友的协议中可看到 XMPP 多么耗费网络流量!

- MQTT：IBM 开发的聊天协议，一个简单的消息协议。
- 类 ActivitySync：微信实现的协议，省流量，性能高，但由于是私有协议，IM 的所有功能都需要自己实现。

对于创业型的公司来说，从上面三种协议的对比可知，如果需要在最短时间内实现聊天功能，除了使用环信、融云等第三方 IM 服务外，最好选择 XMPP 协议。

以下为两个基于 XMPP 协议的著名开源聊天服务器。

（1）Ejobberd：用 Erlang 语言开发，成熟稳定，支持集群，支持多进程、高并发。但由于其是基于小众的 Erlang 语言，造成了很高的开发和维护成本：招聘熟悉 Erlang 同时也熟悉聊天的研发人员不容易。

（2）Openfire：用 Java 语言开发，成熟稳定，插件多，但其对内存要求高，并发低，集群功能弱。

在 App 产品中如果因为某些原因不能使用第三方 IM 服务（例如上级领导不信任第三方 IM 服务），那笔者建议不熟悉 Erlang 语言的开发人员使用 Openfire（如果熟悉 Erlang 语言当然是用 Ejobberd 更好），毕竟熟悉 Java 的开发人员还是挺多，而且在产品初期一般并发量不高，等资金和工程师充足后再对聊天系统进行改造。

某些研发人员有追求完美的天性，但项目初期的环境决定了没法打造完善的系统，使用 Openfire 不是最完美的方案，但最起码能先把聊天功能做出来。

如果项目中必须使用自定义的通信协议，那么开发人员先要解决的是 TCP 协议通信中常见的粘包问题。

App 的通信协议一般是使用 TCP 协议，TCP 协议是一种面向连接、流，提供可靠服务的协议。

当网络上的数据包到达接收端后，程序会根据预先设置的缓冲区大小取数据。在图 9-8 中，假设用户缓冲区为 15 字节，当第 1 个数据包（10 字节）和第 2 个数据包（10 字节）到达接收端后，程序根据缓冲区的大小（15 字节） 取数据，这样缓冲区的数据包含了第 1 个数据包的

全部数据（10 字节）和第 2 个数据包的部分数据（5 字节），当把缓冲区的数据取出，其包含了第 1 个数据包的全部数据和第 2 个数据包的部分数据，于是产生粘包问题。粘包产生原理如图 9-8 所示。

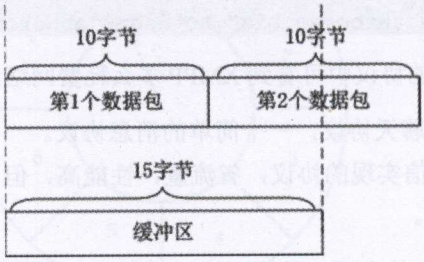


图 9-8　粘包产生原理

例如第 1 个数据包的数据“1234567890”，第 2 个数据包的数据“012345678”，从缓冲区取出全部数据后为“123456789001234”，如果没有明显的标识，程序根本不知道怎么在缓冲区中获取第 1 个包的数据。

解决这个问题的通用方案是制定合理的协议格式，在每个数据包中标明这个包的长度，这样程序就能准确理解和处理包的数据。

笔者下面介绍的两种协议格式，其不但解决了粘包的问题，而且都是高效的通信协议格式。

1. MySQL 协议格式

MySQL 协议中一个数据包的前 3 个字节是数据包长度，第 4 个字节是数据包的序列号，剩下的字节是数据，如图 9-9 所示。

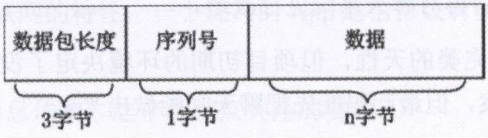


图 9-9　MySQL 协议格式

MySQL 协议通过在数据包头标明了数据包长度，程序就能在缓冲区通过截取正确的数据长度来获取一个数据包。

2. Redis 协议格式

Redis 协议是以 CR LF(CRLF)结尾，协议格式如下。

```
*参数个数 CR LF
$第一个参数的字符串占用字节数 CR LF
```

```
参数数据 CR LF
…
$第 N 个参数的字符串占用字节数字 CR LF
参数数据 CR LF
```

Redis 命令“set name jeff”，根据 Redis 协议可表示如下。

```
*3\r\n$3\r\nset\r\n$4\r\nname\r\n$4\r\njeff\r\n
```

根据公开的资料，陌陌是采用了和 Redis 协议相似的协议作为陌陌的通信协议。

根据 Redis 协议的格式，程序也很容易完整地获取一个数据包的所有数据。

读者如果需要开发私有协议，可以上面介绍的两种协议格式作为模板，打造属于自己的协议。

由于移动互联网的弱网络性，经常会出现丢包的情况（即客户端收不到服务器发送的数据），对于推送、聊天这类应用来说，丢包后有下面两个问题需要处理。

- 怎么确定客户端是否接收到消息？
- 怎么确定需要重发哪些消息？

下面先介绍基于队列的协议在解决上面两个问题上的缺点，再为读者介绍新式的协议是如何有效解决以上的两个问题。

1. 基于队列的消息协议

传统的 IM 协议一般是基于队列的消息发送和反馈机制。

假设服务器中要发送的消息队列如下。

消息队列（先进先出机制）
msg3
msg2
msg1

服务器按照顺序把消息队列中的消息依次发送给客户端，当客户端收到消息后给服务端发送“确认收到”的应答。如果过了一段时间服务器还没收到客户端的应答，就重发该条消息。服务器与客户端之间的消息交互过程如图 9-10 所示。

这种方式有下面两个问题。

- 如果客户端已收到消息并且发送了“确认收到”的应答，但由于网络中断等原因造成服务器收不到应答，这样服务器就会重发该消息，导致客户端收到重复的消息。
- 由于移动互联网的弱网络性，每条消息都需要应答的方式极其费时，服务器要维护每个消息的状态也容易出错。

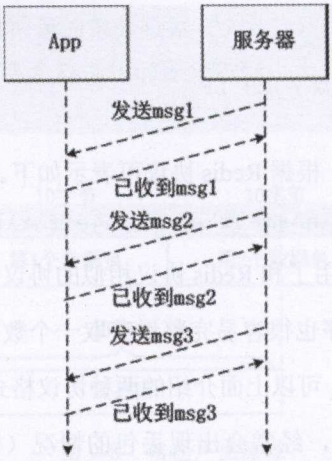


图 9-10　基于队列的服务器和 App 的消息交互过程

这种方式对网络的要求比较高，适用于基于网线、Wi-Fi 等网络信号比较强的情况。

2. 基于版本号的消息协议

服务器上消息按照如下的结构存储，每个消息有一个自增不重复的版本号。

版本号	消息
1	msg1
2	msg2
3	msg3

服务器和 App 之间的消息交互，基于版本号的方式如图 9-11 所示。

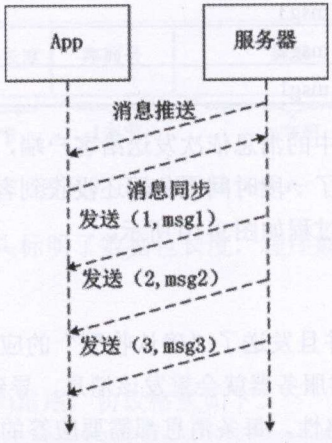


图 9-11　基于版本号的服务器和 App 之间的消息交互

在图 9-11 中，当服务器准备推送消息给 App 时，服务器向 App 发送“消息推送”消息，App 收到这个信息向服务器返回“消息同步”消息（附上最后收到的消息版本号），服务器根据这个版本号，把大于该版本号的所有消息按照版本号依次推送给 App。

假设在图 9-11 中，App 收到消息（2，msg2）后网络中断了，等网络恢复后，App 向服务端发送 “消息同步”信号（默认情况下 App 连上网络后发送这个信号到服务器同步消息），附上最后收到的版本号 2，服务器把版本号大于 2 的所有消息再次推送到 App。

这种交互方式最大的好处是减少了交互的次数，而且依赖客户端维护消息版本号的方式保证服务器上的消息都能同步到客户端，不会丢失消息。

据公开的资料，陌陌和微信采用了类似上面所描述的基于版本号的消息协议。

聊天还有一个常见的功能：发送图片、声音等文件。笔者建议是在聊天中采用纯文本的方案发送这些数据，流程如下。

（1）假设用户 a 向用户 b 发送了一张图片，发送前 App 先把该图片上传到文件服务器后得到一个可访问的 URL：http://file.test.com/1.jpg。

（2）App 聊天模块对接收到的下面几种文本类型的数据做不同的处理，如下所示。

文本类型	展现的数据	处理方案
{"type":"txt","content":"hello"}	文本	在 App 中直接显示文本内容
{"type":"image","content":"http://file.test.com/1.jpg"}	图片	先把图片下载，再在 App 中显示
{"type":"sound","content":"http://file.test.com/1.amr"}	语音	先把语音下载，再在 App 中播放

发送图片声音等文件流程如图 9-12 所示。

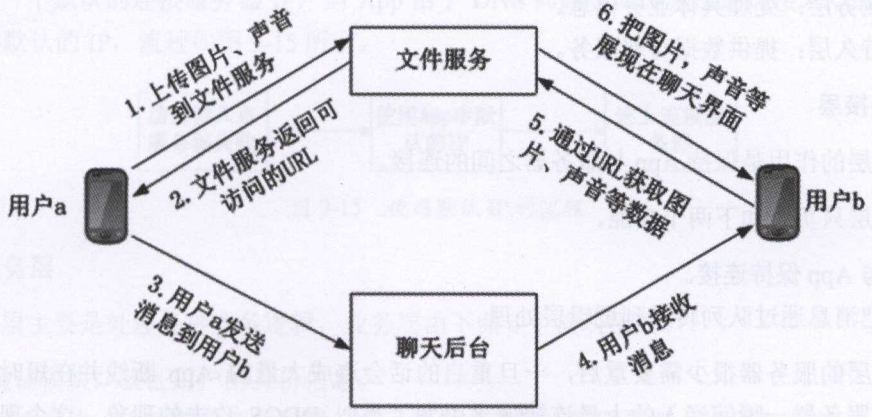


图 9-12　发送图片声音等文件流程

采用纯文本传输图片、声音等消息的好处如下。

（1）无论是 App 端或者是聊天后台，只处理文本的话都比较简单。

（2）把文件放在文件服务中，可以统一优化文件的性能（例如使用文件云存储或者 CDN），不需要单独优化聊天相关的文件。

9.1.3 整体架构

聊天 App 后台架构如图 9-13 所示。

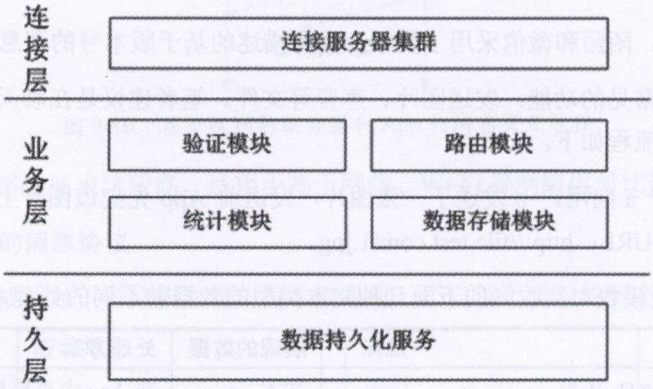


图 9-13　聊天 App 后台架构

聊天 App 后台架构由下面 3 个部分组成。

- 连接层：集群包含数量众多的连接服务器。
- 业务层：处理具体业务信息。
- 持久层：提供数据存取服务。

1. 连接层

连接层的作用是保持 App 与服务器之间的连接。

连接层只负责如下两个功能。

- 与 App 保持连接。
- 把消息通过队列转发到逻辑层处理。

连接层的服务器很少需要重启，一旦重启的话会造成大量的 App 断线并在短时间内重连服务器，服务器一瞬间涌入的大量连接请求引发了类似 DDOS 攻击的现象。这个现象可以通过合理的机制杜绝：当连接层的服务器准备重启，发送一条消息到连接着这台服务器的 App，

让其在重连时连接别的服务器，不要连接当前的服务器。

连接层有一个常见的问题：连接层是由多台服务器组成的集群，那么某个 App 要连接哪台服务器？解决方案是动态分配接入点，这个方案如下。

App 访问接入点服务器，接入点服务器根据各个连接服务器的负载等因素综合计算，返回一个连接服务器的 IP 给 App，App 通过这个 IP 连上连接服务器，整个流程如图 9-14 所示。

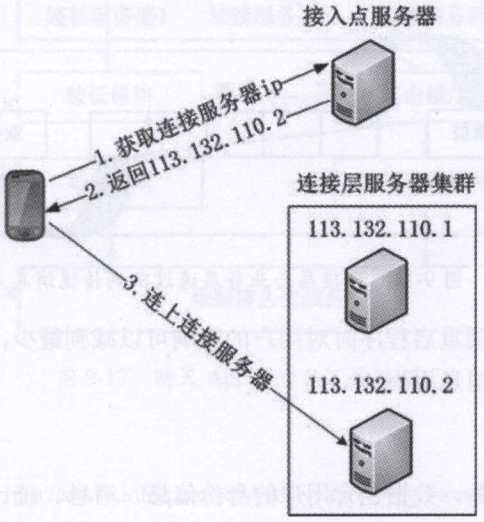


图 9-14　动态分配接入点

如果因为 DNS 故障等原因无法通过接入点服务器获取 IP，可以使用预埋 IP 的方案：在 App 中有一个默认的连接服务器 IP，当 App 由于 DNS 问题等原因无法通过接入点服务器获取 IP 就连接默认的 IP，流程如图 9-15 所示。

图 9-15　使用默认 IP 的流程

2. 业务层

业务层主要是处理各种业务逻辑，业务层由下面 4 个模块组成。

- 验证模块：验证用户的身份信息。
- 路由模块：连接服务器的集群包含了数量众多的连接服务器，例如当 A 用户向 B 用户发送消息，两个用户的连接不一定是在同一个服务器，因此需要通过路由模块获取用

户所在的服务器。如果要实现群聊功能，还要在这个模块中查找“订阅/发布”关系。

- 统计模块：统计各种信息，例如总连接数、每秒发送消息数、总用户数、Android 客户端连接数、iOS 客户端连接数等等。
- 数据存储模块：存储消息，统计信息，用户身份信息等。

连接层通过队列向业务层进行消息传递，业务层不断从队列中取出消息进行相关的处理，如图 9-16 所示。

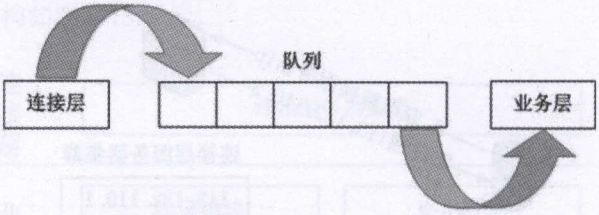


图 9-16　连接层与业务层通过队列传递消息

使用了队列后，业务层重启程序时对用户的影响可以减到最少，因为所要处理的消息都保存在队列，重启不会造成消息丢失。

3. 持久层

该层提供数据存取服务，数据包含用户的身份信息、消息、统计信息等。

根据不同数据对存取速度不同的要求，可使用不同的软件存储不同的业务数据。例如，用户身份信息这种高频读写的数据，可存储在 Redis、Memcached 等内存数据库中。

4. 工作流程

下面以用户 1 向用户 2 发送消息“你好”为例，说明聊天 App 后台的工作流程。

（1）用户 1 向连接服务器 1 发送消息（里面包含了接收的对象用户 2 的唯一标识和内容“你好”）。

（2）连接服务器 1 接收到该消息后向用户 1 返回一个应答，同时通过队列把消息传递到业务层。

（3）业务层的验证模块先验证该用户的身份信息，验证完成后通过路由模块确定用户 2 所在的服务器为连接服务器 2，把消息传递到连接服务器 2 从而发送到用户 2 的手机上。

（4）业务层把消息传递给用户 2 时，对相关的消息进行统计，同时通过数据存储模块把数据持久化。

聊天 App 后台的工作流程如图 9-17 所示。

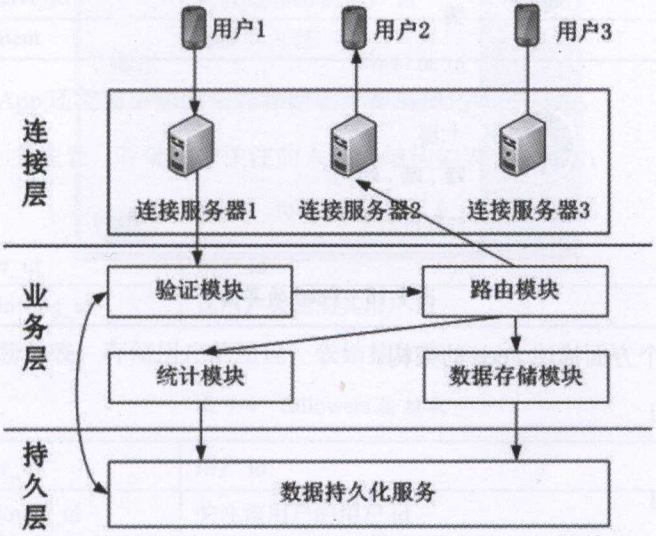


图 9-17　聊天 App 后台的工作流程

参考资料：

（1）讲师：李志威 http://www.infoq.com/cn/presentations/high-availability-instant-communication-architecture《高可用即时通信架构》

（2）《微信技术总监周颢：一亿用户背后架构秘密》http://tech.qq.com/a/20120515/000224.htm

9.2　社交 App 后台架构

社交 App 最常见的场景是类似于微博的场景，用户与用户之间有关注和粉丝这两种关系，一个用户发表了内容，关注了该用户的用户在个人主页上都能收到最新的动态。

社交的核心功能是 Feed。 Feed 是指用户通过关注功能，聚合了被关注用户的最新的内容（同时也包括了自己的内容）以供自己浏览的信息服务。

Feed 的界面如图 9-18 所示，按发表的时间顺序把关注的用户的内容展现出来。

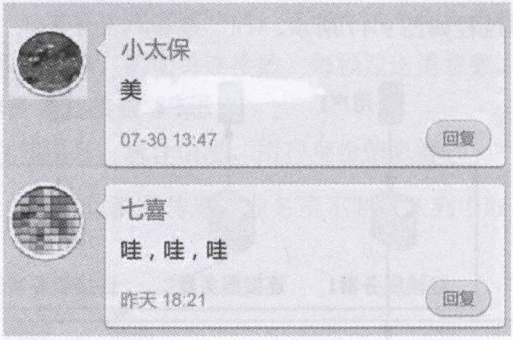


图 9-18　Feed 的界面

本节从下面 4 个方面描述 Feed 的架构。

- 基本表结构
- 推拉模式
- 数据库架构
- 缓存架构

9.2.1　基本表结构

常用的 Feed 架构是把数据存储在 MySQL，热点数据（一般来说是最近 3 天的数据）存储在缓存（常见的缓存系统有 Redis 和 Memcached，微博是使用 Memcached），务必让绝大多数请求通过缓存直接返回（例如微博要求核心缓存命中率要达到 99%），只有少量的请求穿透缓存落到数据库。

在最简单的 Feed 架构中，MySQL 有下面的 4 张基本表。

- send_content：发送内容表，存储用户发表的内容，表结构如表 9-1 所示。

表 9-1　send_content 发送内容表

feed_id	发表的 feed 的 id
author_id	发表该 feed 的用户 id
content	feed 的内容

- reveive_content：接收内容表，用于推模式（推模式的描述请看下节）时存储用户接收的内容，表结构如表 9-2 所示。

表 9-2　reveive_content 接收内容表

feed_id	发表的 feed 的 id
author_id	发表该 feed 的用户 id

续表

receive_id	接收该 feed 的用户 id
content	feed 的内容

另外，在社交 App 还需要下面两张表存储用户和用户之间的关系。

- followings：关注表，存储用户关注的人，表结构如表 9-3 所示：

表 9-3　followings 关注表

user_id	用户 id
following_id	该用户关注的其用户 id

- followers：粉丝表，存储用户的粉丝，表结构如表 9-4 所示：

表 9-4　followers 粉丝表

user_id	用户 id
follower_id	关注该用户的用户 id

9.2.2　推拉模式

用户发表内容后，后台通过一定的方式把获取数据展示到该用户的粉丝主页，常用的两种方式是推模式和拉模式。

1. 推模式

平常邮箱中收件箱和发件箱的作用如下。

- 发件箱：存储用户发送的邮件。
- 收件箱：存储用户收到的邮件。

推模式的数据库设计也有收件箱和发件箱类似的概念，发件箱对应表 9-1“send_content 发送内容表”，收件箱对应表 9-2“reveive_content 接收内容表”。

推模式下用户发表一条内容的流程如下。

（1）id 为 1 用户发表一条内容为“蓝蓝的天空”信息。

（2）这条内容写入发送内容表“send_content”后内容如下。

feed_id	author_id	content
1	1	蓝蓝的天空

（3）在粉丝表“followers”查找 id 为 1 用户的粉丝，粉丝表“followers”的内容如下。

user_id	follower_id
1	2

从粉丝表可知，id 为 1 用户的粉丝是 id 为 2 的用户。

（4）因为 id 为 2 的用户的 feed 中需要显示这条内容，因此把内容写入接收内容表“reveive_content”，写入后接收内容表“reveive_content”内容如下。

feed_id	author_id	receive_id	content
1	1	2	蓝蓝的天空

（5）当 id 为 2 的用户显示 Feed 时，通过 SQL 语句“select * from reveive_content where revcive_id=2”就能查询该用户需要显示的数据。

从推模式的流程可看出，推模式使用了空间换时间的策略，查找 Feed 中需要显示的内容很简单，只需要一条 SQL 语句就行。

同样推模式的缺点也很明显，如下所示。

- 同时推给多人（例如上十万人）不但延时严重，而且浪费存储空间。
- 变更操作的成本高，如果某个用户删了一条内容，要同时把“send_content”表和“reveive_content”表中的数据删除。

2. 拉模式

拉模式下用户发表一条内容的流程如下。

（1）id 为 1 用户发表了一条内容，内容为“蓝蓝的天空”。

（2）这条内容写入发送内容表“send_content”，写入后发送内容表“send_content”内容如下。

feed_id	author_id	content
1	1	蓝蓝的天空

（3）当 id 为 2 的用户显示 Feed 时，在关注表“followings”查找 id 为 2 所关注的用户，关注表的内容如下。

user_id	following_id
2	1

从关注表“following_id”可知，id 为 2 的用户关注了 id 为 1 的用户，因此需要获取 id 为 1 的用户发表的内容。

（4）id 为 2 的用户通过 SQL 语句“select * from send_content where author_id in (1)”查询

所有需要显示的内容。

由上述流程可知，拉模式采用了时间换空间的策略，用户推送内容时效率很高，但当用户显示 Feed 时，需要花大量的时间在聚合运算上。

3. 推拉模式的总结

推模式和拉模式的特点总结如下。

	推模式	拉模式
发表内容	推送给所有粉丝	不推送
显示 Feed	一个 SQL 语句就能完成	需要大量的聚合运算
变更通知	变更成本高	没有变更成本

4. 微博的推拉模式

新浪微博中公开的微博采用拉模式，私密性的微博采用推模式。

读者通过前面的介绍可知，拉模式最大的问题是大量的聚合运算，请求的响应时间可能较长，可以通过缓存策略让大部分的请求的响应时间能达到 2 到 3 毫秒。关于缓存的架构，请查阅本章“9.2.4 缓存架构的演进”。

注意：本节中的 Feed 所使用的推拉模式默认为不显示该用户发表的内容，读者如果需要显示用户自身发表的内容，在上面的流程上适当改进即可。

9.2.3　数据库架构的演进

一般数据库架构的演进过程如下：

- 单机部署。
- 读写分离，从一主一从到一主多从。
- 分表分库。

社交 App 后台数据库架构的演进也是类似的，前两个阶段是比较通用（读者可查看本书“6.6 架构演化”），在分表分库阶段除了使用在“6.6.3 分库”中介绍的数据库分布式处理软件 MyCAT 外，还能用业务层的分表分库策略。读者可能会有疑惑，为啥不用数据库分布式处理软件而要在业务层实现分表分库？因为在业务层实现分表分库可以获得更大的灵活性，例如可以实现冷热数据的分离存储：最近的数据存储在高性能的设备上，旧的历史数据存储在廉价的设备以节省成本。

下面介绍社交 App 后台数据库在业务层实现分表分库的方案。数据库的分表分库，首先要解决的是数据库自增 id 问题。

1. 数据库自增 id

数据库分表分库后，程序聚合了多张表的数据时有可能出现的问题是出现重复的 id。在实现分表分库前，id 是采用在表中自增的策略，当表分散后就不能保证 id 的唯一性。为了保证分表后 id 的唯一而且自增，需要一个全局统一的发 id 号服务。

假设业务上每秒峰值写入是 100 万，如果采用全局统一的发 id 号服务，就是要区分每秒的 100 万条数据。多数互联网公司都会有自己的发 id 号算法，常见使用的是 time+sequeue 设计方式。例如，设计中 id 的长度为 64 位，则 id 的前 32 位（可表示数值 0~4294967295）精确到秒（用时间戳表示），中间 20 位（可表示数值 0~1048575）就可以表示 0~100 万之间的任意整数（即 sequeue），最后 12 位存放其他业务信息。

2.分表分库策略

常见的分表分库策略有下面两种。

- 按 hash 拆分。
- 按时间拆分。

（1）按 hash 拆分

按照特定的 hash 算法，把一张表的数据分布到多张表。例如，把内容表分为 4 张表，根据发表内容的用户 id 做 hash 运算，用算法 id%4 就能计算出这条数据落在哪张表上，如图 9-19 所示。

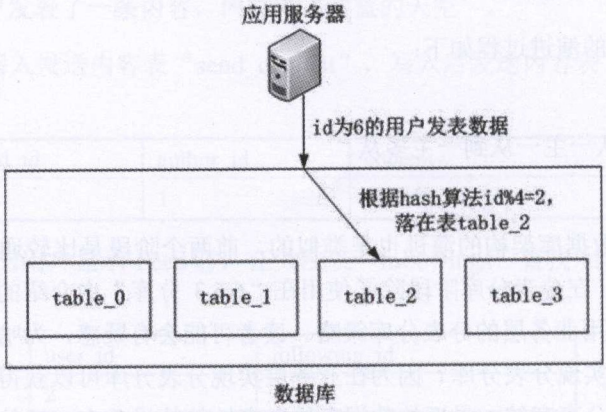


图 9-19　按 hash 拆分数据

按 hash 拆分是最常见、最简单的分表分库方案，适合于分表分库的前中期使用，在这个阶段数据库的数据通常有下面的特点。

- 数据规模可控。
- 增长速度可控。

hash 拆分有下面的缺点。

- 特殊用户的查询性能低。如果某些用户数据特别多，按照用 id 值 hash 策略会造成数据集中在某个数据表，造成查询和写入操作都集中在某张表上。
- 冷热数据没法分离存取。例如某些热点数据可以放在价格高、读写性能强的硬盘上提高读写速度，冷门的数据可放在价格低、读写性能稍弱的硬盘。对于社交类的 App 来说，热门数据一般就是用户最近发表的内容，按照用户 id 值，hash 分表没法满足这个需求。

采用下面介绍的按时间拆分的策略，可以弥补 hash 分表的缺点。

（2）按时间拆分

按时间拆分的策略是将同一时间段的数据放在同一张表，不同时间段的数据放在不同的表。例如，2015 年 9 月份发表的内容记录在表 “table_201509”。时间拆分如图 9-20 所示。

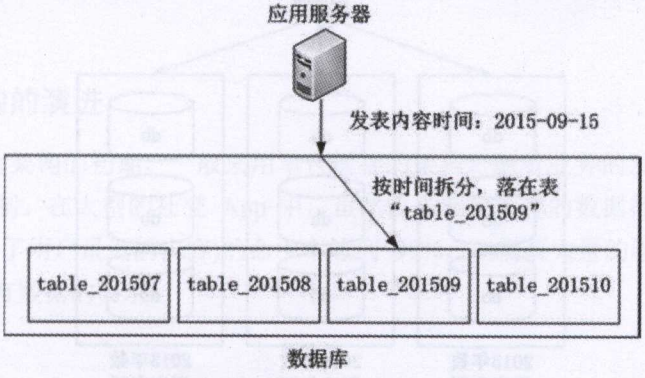


图 9-20　时间拆分示意图

（3）综合的策略

分表分库中可以综合使用 hash 和时间两种分表分库策略：用户发表内容，先按照用户的 id 的值 hash 到具体的库，再按照时间落到具体的表，如图 9-21 所示。

在图 9-21 中实现的是一种数据库内的冷热数据分离策略，还有一种更大范围内的分表分库方案可以实现冷热数据分离，先按照年份选定一个数据库集群，在集群内部按照用户 id hash 到具体的库，再按照月份确定到具体的表，如图 9-22 所示。

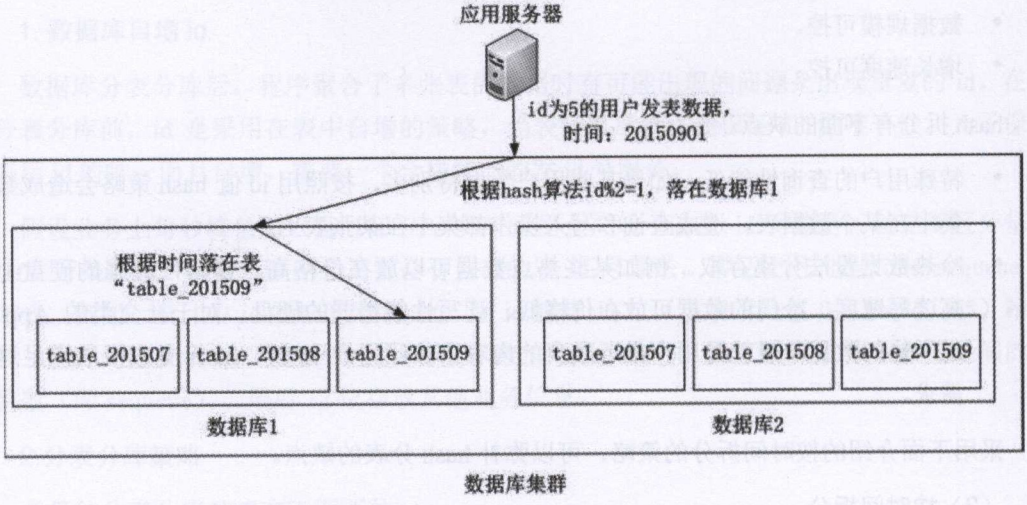


图 9-21　综合使用 hash 和时间两种分表分库图

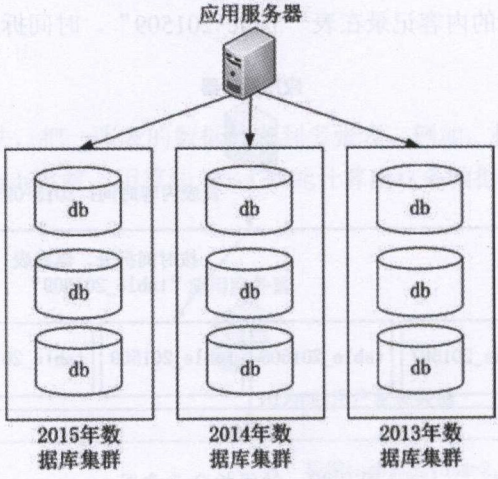


图 9-22　更大范围的分表分库策略

按照时间拆分 Feed 数据后需要解决一个问题：因为社交类 App 常见的业务场景是显示用户最近发表的内容，怎么快速查找用户最近发表的内容？例如这样的需求：快速找到 id 为 1001 的用户最近发表的 5 条内容，难道要把图 9-20 中的 4 张表都查找一遍吗？这样效率太低了。

解决这个问题需要引入一个二级索引表，通过这个二级索引表查找到具体的数据，如图 9-23 所示。

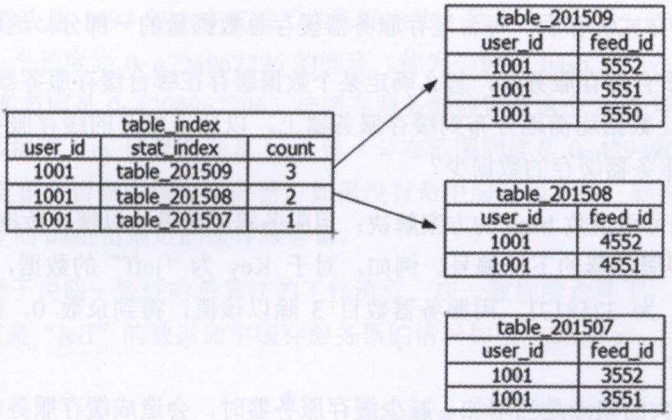


图 9-23　二级索引示意图

例如需要查找 id 为 1001 的用户的最近发表的 5 条数据，按图 9-23 的表结构，可在二级索引表“table_index”获知有 3 条数据在索引表“table_201509”，有 2 条数据在索引表“table_201508”。通过索引表“table_201509”查到 3 条数据 的 feed_id 为 5552、5551、5550，通过索引表“table_201508”查到 2 条数据 的 id 为 4552、4551，根据查找到的 id 很容易就能获取到详细的数据。

9.2.4　缓存架构的演进

社交 App 后台架构的初期，一般采用单台缓存的架构，随着业务的发展，单缓存的架构会受到单机内存限制。在大型的社交 App 中，虽然没必要把所有的数据都放入缓存，但为了保证核心缓存(保存了用户最近的内容)的命中率达到 99%，也需要大量的缓存。单台服务器显然没法满足缓存所有数据的要求。

1. 分布式缓存

为了解决单机内存受限的问题，社交 App 后台引入了分布式缓存的架构，如图 9-24 所示。

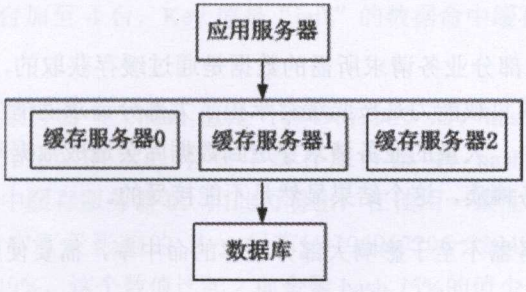


图 9-24　分布式缓存的架构

后台使用了分布式缓存后，每台缓存服务器缓存总数据量的一部分，这时要解决两个问题。

- 后台中有多台缓存服务器，怎么确定某个数据缓存在哪台缓存服务器？
- 怎样才能让数据均衡地分布到缓存服务器上，以免造成有的缓存服务器缓存的数据多，有的缓存服务器缓存的数据少？

上面两个问题可用余数 hash 的方案解决：用服务器的数量除以缓存数据的 Key 的 hash 值，余数为对应的缓存服务器的下标编号。例如，对于 Key 为“jeff”的数据，hash 值（Java 的 hashCode 返回值）为 3258171，用服务器数目 3 除以该值，得到余数 0，因此该数据存储在“缓存服务器 0”。

余数 hash 最大的缺点是当增加、减少缓存服务器时，会造成缓存服务器结果的大震荡。例如，缓存服务器从 3 台变成了 4 台，对于 Key 为“jeff”的数据，hash 值（Java 的 hashCode 返回值）为 3258171，用服务器数目 4 除以该值，得到余数 3，因此把数据从存储在“缓存服务器 0”变为存储在“缓存服务器 3”，如图 9-25 所示。

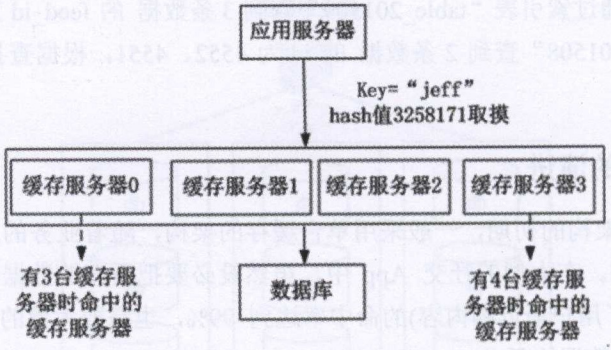


图 9-25　Key 为“jeff”的数据命中缓存服务器的不同情况

读者可以计算出来，当缓存服务器的数目从 3 台升到 4 台，大约有 75%（3/4）被缓存的数据不能命中缓存服务器，随着服务器集群规模的增大，这个比值会不断加大，整体的命中率下降得非常快。

在 App 后台中，大部分业务请求所需的数据是通过缓存获取的，只有少部分请求会穿透到数据库，因此数据库的负载能力是按照缓存承担了大部分请求的情况设计。当因为增加缓存服务器造成缓存没法命中，大量的业务请求穿透到数据库会造成数据库的压力过大，甚至造成数据库宕机，从而使业务瘫痪，这个结果显然是不能接受的。

为了使新增缓存服务器不至于影响大部分缓存的命中率，需要使用“一致性 hash 算法”这种更合理的 hash 算法。

一致性哈希算法是 1997 年由麻省理工学院提出的一种分布式哈希（DHT）实现算法。其基本原理是构造一个长度为 0~4294967296 的圆环（称为一致性 hash 环），根据缓存服务器名称的 hash 值（分布范围是 0~4294967296）把缓存服务器放置在这个 hash 环上。当缓存的请求到达，根据缓存数据的 Key 计算得到其 hash 值（分布范围同样是 0~4294967296），根据这个 hash 值判断缓存数据是否命中缓存服务器，如果没有命中缓存服务器，则顺时针在 hash 环上查找距离这个 Key 的 hash 值最近的缓存服务器。

下面举一个例子说明一致性哈希算法的工作流程。在一致性哈希环下，存在 3 台缓存服务器的情况，Key 值是“jeff”的数据命中缓存服务器的情况如图 9-26 所示。

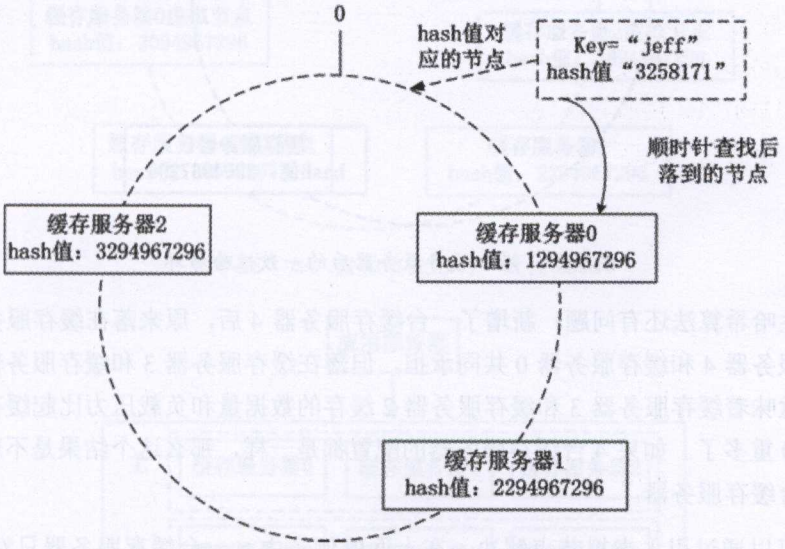


图 9-26　原始的一致性哈希环

按照一致性哈希算法（顺时针查找的特性），在只有 3 台缓存服务器的时候，hash 值范围是（3294967296~4294967296，0~1294967296）的 Key 都会命中缓存服务器 0。

当缓存服务器从 3 台加至 4 台，Key 值是“jeff”的数据命中缓存服务器的情况如图 9-27 所示。

当增加了一台缓存服务器 4（hash 值是 1094967296）后，hash 值范围是 3294967296~4294967296，0~1094967296）的 Key 会命中缓存服务器 4，hash 值范围是 1094967297~1294967296 的 Key 会命中缓存服务器 0。因此可看出，在使用一致性哈希算法的情况下，增加了一台缓存服务器后命中率受影响的 Key 只有[(4294967296-3294967296)+(1094967296-0)]/4294967296=0.2549=25.49%，这个数值比起之前余数 hash 75%的值少多了。

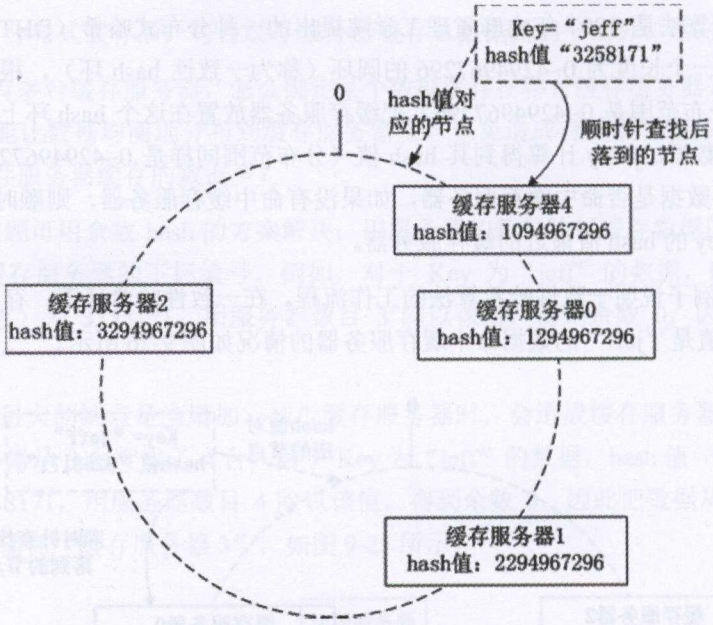


图 9-27　增加缓存服务器后的一致性哈希环

但一致性哈希算法还有问题：新增了一台缓存服务器 4 后，原来落在缓存服务器 0 的请求现在由缓存服务器 4 和缓存服务器 0 共同承担，但落在缓存服务器 3 和缓存服务器 2 上的请求不变，这就意味着缓存服务器 3 和缓存服务器 2 缓存的数据量和负载压力比起缓存服务器 4 和缓存服务器 0 重多了。如果 4 台缓存服务器的配置都是一样，那么这个结果是不理想的，没法充分利用每台缓存服务器。

该问题可以通过引入虚拟节点解决。在上面的例子中，一台缓存服务器只对应一个 hash 值，如果把一台缓存服务器对应两个 hash 值（一个真实节点，一个虚拟节点），甚至是多个 hash 值（多个虚拟节点）呢？这样新加入一台缓存服务器，将会均匀地影响已经存在的缓存服务器并分摊原来服务器集群中的负载。新的一致性哈希算法环如图 9-28 所示。

2. 主从缓存结构

分布式缓存使用了一致性 hash 算法后，缓存还会存在一个问题：社交 App 后台对核心缓存的命中率要求极高，就算使用了一致性 hash 算法，当其中一台缓存服务器宕机后，也会造成缓存命中率的大幅下降。

解决上面的问题需要引入主从缓存结构的分布式缓存，如图 9-29 所示。

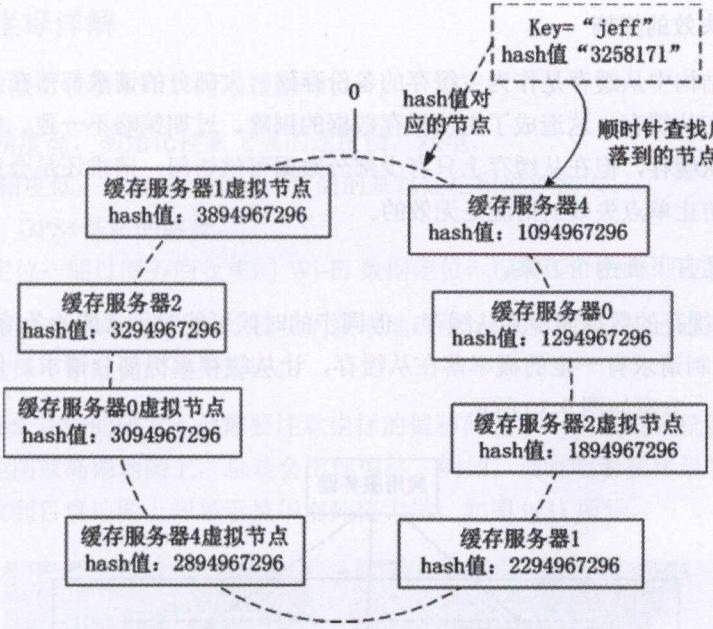


图 9-28　加入虚拟节点后的一致性 hash

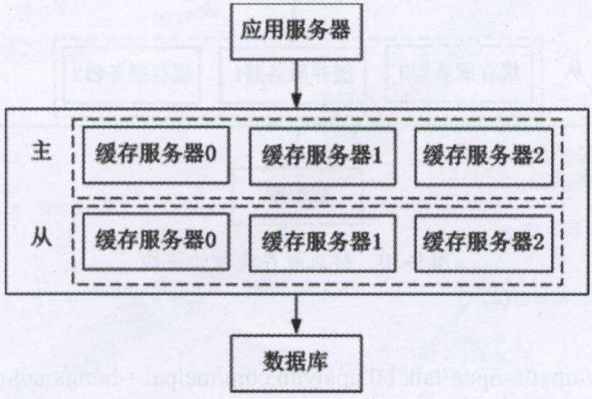


图 9-29　主从缓存结构的分布式缓存

获取数据的时候，先访问主缓存，当主缓存获取数据失败（例如服务器宕机等原因）后，再访问从缓存。

在两层缓存结构中数据以主缓存为主。当更新数据的时候，先从主缓存获取数据，再对主缓存进行一致性更新，更新成功后再更新从缓存，如果主缓存一致性更新多次都失败，则把主缓存、从缓存的数据删除，后续的请求穿透到数据库获取数据后回写到主、从缓存。

3. 防止缓存失效的措施

主、从缓存结构中从缓存是作为主缓存的备份存储。大部分的请求都落在主缓存，只有少部分的请求会落在从缓存，这造成了主从缓存数据的剔除、过期策略不一致。如果主缓存出了问题，请求落在从缓存，但在从缓存上只有少部分数据可供访问，请求还是会穿透到数据库，因此从缓存作为防止单点失效的措施是无效的。

解决这个问题有下面两个方案。

- 定期把主缓存的数据同步到从缓存，但同步的时候可能对正常的业务请求有影响。
- 应用层控制请求有一定的概率落在从缓存，让从缓存承担部分请求，使从缓存中的数据不过冷。方案如图 9-30 所示。

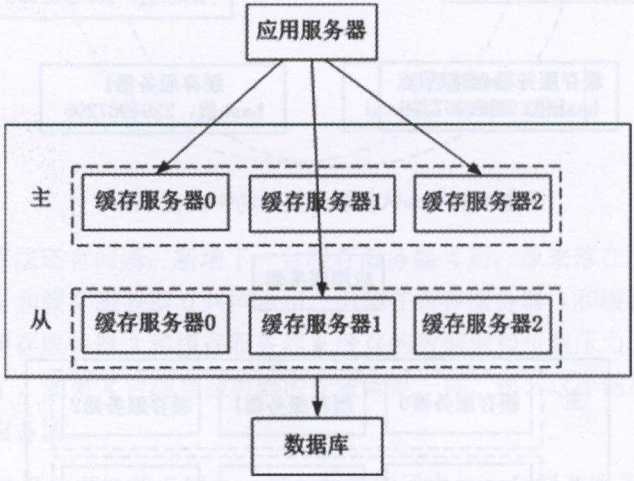


图 9-30　防止缓存失效的架构

参考资料：

讲师洪小军 http://upyun-open-talk.b0.upaiyun.com/meipai---hongxiaoj.mp4《大中型 SNS 系统设计漫谈》

9.3　LBS App 后台架构

移动互联网除了一直在线这个特点外，还有一个重要特点：手机可以定位用户的位置。查找附近的人、餐馆等服务，以及大量的 O2O 应用，都需要使用 LBS(Location Based Services)。本节介绍 LBS App 后台架构中相关的知识。

9.3.1　地理坐标详解

下面 4 种方法可以获取用户的地理坐标。

- GPS：精度高，初始化搜索卫星的速度慢，耗电。
- 基站：精度低，速度快，不同运营商的基站定位精度差别大。
- AGPS：GPS+基站的结合。
- Wi-Fi 定位：通过服务商收集的 Wi-Fi 数据定位，但 Wi-Fi 的地理位置信息更新非常慢。

App 端建议直接使用地图 SDK 提供的获取地理坐标的方法来获取地理坐标，其会综合各种定位方式后选择一个最优的结果返回。

初次做 LBS 功能的研发人员需要注意坐标的偏移问题。App 通过系统级的函数获取的坐标，放到百度地图或高德地图上，总是会出现偏移。例如，当时笔者在华景软件园附近获取的坐标，把坐标放到百度地图上却显示是华南师范大学，如图 9-31 所示。

图 9-31　坐标的偏移情况

经过查找资料，笔者终于知道问题所在，如下。

通过系统底层函数获取的坐标是国际坐标系 WGS-84(World GEodetic System 1984)，是为 GPS 全球定位系统的使用而建立的坐标系统。

GCJ-02 是由中国国家测绘局制订的地理信息系统的坐标系统，其是对经纬度数据（WGS-84）的加密算法，即加入随机的偏差。对 WGS-84 获取的坐标进行一次加密偏移。国内出版的各种地图数据必须至少采用 GCJ-02 对地理位置进行首次加密。假设 GPS 获取的坐标是（113.37，23.04），这个坐标在国内的地图上经过偏移就可能变成（113.39，23.06）。而且这个偏移量没有明确规定，这就造成了一个现象，不同电子地图服务提供商有不同的坐标体系，例如，Google 地图、高德地图是同一套坐标体系，百度地图却是另外一套坐标体系。

解决这个问题的方案：使用百度地图 SDK 提供的获取地理坐标功能，其获取的坐标是已经偏移过的，这个坐标在百度地图上能准确显示。

9.3.2 查找附近的人

LBS 主要用于查找附近的人（或者某个事物），其常见的使用情景如图 9-32 所示。

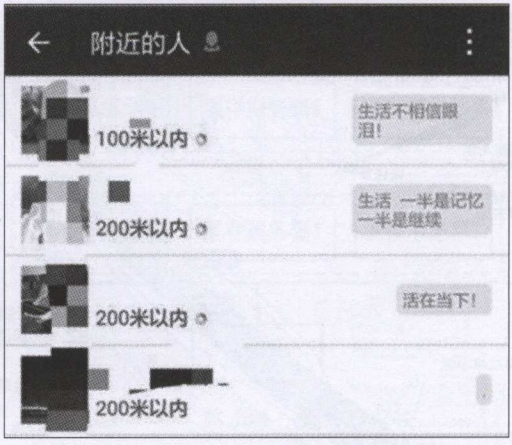


图 9-32 查找附近的人

查找附近的人在程序中的处理流程是：数据库的用户信息中有一个地理坐标字段，已知某个用户的地理坐标，把这个坐标一定范围内（例如 500 米内）的其他地理坐标找出来。

下面介绍 3 种实现查找附近的人的方案。

1. MySQL 的空间数据库

从 MySQL 4.1 开始就引入了一系列空间扩展，使其具备了一定的空间处理能力。

简单点来说，就是 MySQL 已经可以把地理坐标的数据当成一种独立的数据类型，而且提供了相关的空间函数（例如查找一个矩形范围内的坐标）给开发者直接调用。

2. geohash

geohash 编码，就是通过算法把地理坐标转换成一个值，简单点来说就是把二维坐标转换成一个字符串，但这个字符串并不是毫无意义。

geohash 有以下的特点。

（1）geohash 把地理坐标转换为字符串。

如图 9-33 所示，展示了“国家数字家庭应用示范产业基地”的地图。

图 9-33　国家数字家庭应用示范产业基地

在上图中，“国家数字家庭应用示范产业基地”从地图上得到的坐标是“113.39175，23.061784”，这个坐标同时也可以用 geohash 值“ws0ehq32ek0u”表示。用 geohash 值表示坐标有以下两个好处。

- 避免泄露用户的真实坐标，保护隐私（虽然中国用户并不看重这个问题）。
- geohash 值是个字符串，方便缓存。在下面会提到一个案例用缓存 geohash 值来快速搜索。如果使用坐标值就不方便缓存。

（2）坐标的 geohash 值越相似，意味着距离越相近。

（3）geohash 值越长，表示的范围越精确。如表 9-5 所示，展示了 geohash 值的长度和相应的表示范围的关系 。

表 9-5　geohash 值的长度和相应的表示范围的关系
（数据来源：http://en.wikipedia.org/ wiki/Geohash）

geohash length	lat bits	lng bits	lat error	lng error	km error
1	2	3	±23	±23	±2500
geohash length	lat bits	lng bits	lat error	lng error	km error
2	5	5	± 2.8	± 5.6	±630
3	7	8	± 0.70	± 0.7	±78
4	10	10	± 0.087	± 0.18	±20
5	12	13	± 0.022	± 0.022	±2.4
6	15	15	± 0.0027	± 0.0055	±0.61
7	17	18	±0.00068	±0.00068	±0.076
8	20	20	±0.000085	±0.00017	±0.019

从表 9-5 可看出，当 geohash 值的长度为 6 时，对应坐标的距离大约是相距 0.61 千米，“查找附近的人”这个功能平常就查找 1 公里范围内，按照 geohash 值的长度为 6 查找就差不多。

但是使用 geohash 查找附近的人需要注意如图 9-34 所示的例子。

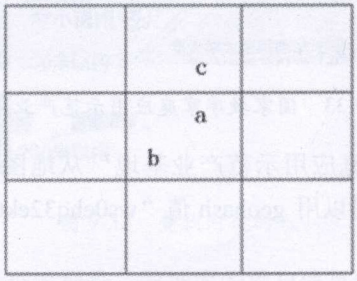


图 9-34　geohash 查找附近的人

在图 9-34 中用 geohash 查找 a 点附近的点时，由于 a 和 b 是在同一个区域内，根据 geohash 算法认为 a 附近只有 b。但在图 9-34 上可看到，c 点到 a 点的距离和 b 点到 a 点的距离差不多，为了取得更精确的结果，除了取目标点的 geohash 值外，还需要使用 a 点周围 8 个区域的 geohash 值。

基于 geohash 值是字符串的特性，查找附近坐标的时候用 SQL 中模糊查询 LIKE “ws0ehh%”查找 geohash 值“ws0ehh”附近的所有地点，非常方便。

笔者早期做过的产品中有一个需求是查找用户附近的商铺(包含关键字)，商铺的数据有130 多万，全放存放在 MySQL 中，发现用 MySQL LIKE“ws0ehh%”这种方式检索 geohash 值性能瓶颈很大，检索 130 万行的数据，平均花费了 8 秒，这种响应速度在生产环境中是无法忍受的。

后来经过不断优化，确定了 CoreSeek+Redis+MySQL 的解决方案，一下子就把响应速度减少到可接受的程度，这个方案的如下。

- 用每个商铺的坐标值转换为 geohash，用 geohash 作为 Key，商铺的 id 作为 Value，存储在 Redis 的 set 结合中。
- 根据用户的坐标计算 geohash，在 Redis 中用“keys*”的方法匹配查找用户附近的商铺 geohash(geohash 的特点是 geohash 编码的前缀可以表示一个区域)，获得商铺 id。例如，用户坐标的 geohash 值是“ws0ehh”，在 Redis 中使用“ws0ehh*”搜索前缀为“ws0ehh”的 Key，根据 Key 获取 set 集合中保存的商铺 id。
- 把商铺 id 作为附加筛选条件，再加上主条件（例如商品的名称关键字）在 CoreSeek 中继续查找相应的数据。

注意：这个方案的性能最终受限于 Redis 的读取速度，由于 Redis 的“*”号操作是遍历 Redis 中的所有 Key，当 Redis 中的 Key 到达一定的规模后响应速度会变慢，现在更推荐使用 MongoDB 来解决 LBS 问题。

3. MongoDB

MongoDB 的一大亮点是封装了大量的地理位置操作，全球流行的 LBS 服务“Foursquare”、国内著名的打车 App“快的”也曾经使用 MongoDB 的地理位置查询功能。

关于 MongoDB 处理 LBS 的原理和常用操作，请查阅章节“8.5 LBS”。

使用 MongoDB 开发 LBS 服务有以下的优点。

- MongoDB 自身的性能高，更新、查询的速度快。
- 通过副本集、分片等方法，很容易实现 MongoDB 的分布式部署，解决性能瓶颈。
- MongoDB 已在 App 后台中广泛使用，很多开发者对开发部署 MongoDB 比较熟悉，减少运维成本。

MongoDB 可支持下面的地理位置查找。

- 查询多边形范围的坐标。
- 查询附近的坐标。
- 查询圆形区域内的坐标。

关于 MongoDB 的 LBS 操作，请读者查看本书“8.5 LBS——地理位置查询”。下面讲述基于 MongoDB 的 LBS 后台架构演进。

9.3.3 基于 MongoDB 的 LBS 后台架构演进

1. 副本集架构

使用 MongoDB 处理 LBS 查询，为了保证高可用，开始阶段就推荐使用 MongoDB 的副本集架构。副本集架简单来说类似于 MySQL 的主从架构，一个主节点负责写，多个从节点负责读，通过 MongoDB 的内部机制，数据从主节点复制到从节点。但副本集中的主节点不是固定于某台服务器，而是通过集群中的服务器选举得到。关于副本集的详细介绍，请读者查看本书“8.4.2 副本集”。

副本集的架构如图 9-35 所示。

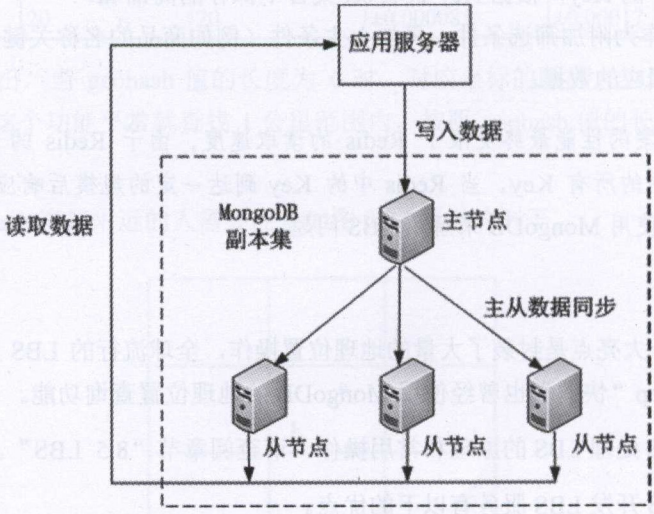


图 9-35 MongoDB 副本集架构

随着业务的发展，MongoDB 副本集慢慢会出现下面的问题。

- 内存不够。
- 读写压力过大。
- 主、从复制延时过长。

不同类型的 App 对 LBS 系统的读写压力完全不同。例如，对于美甲、家政类型的 O2O App，其更新和获取 LBS 数据的频率很低。但对于打车类 App，因为其需要频繁地更新地理位置数据，LBS 后台需要承担的读写压力比一般 App 的读写 App 大，在快的公开的资料中 LBS

系统每秒的读写次数比居然达到 4:1。

MongoDB 写压力增大引发库锁问题。MongoDB3.0 之前的版本会有库锁，库锁的意思是当多个客户端并发访问一个库时，如果某个客户端正在进行写操作，其他客户端都必须排队等待。到了一定的并发量后，库锁对性能影响十分巨大。

MongoDB 写压力增大也会引发主从复制延时过长的问题，对于某些类型的 App 来说是没法接受的，例如打车 App。因为车子一直都处于高速移动，司机的地理位置数据会在主节点更新。当乘客在从节点中查询附近的车子时，如果主、从延时过大，虽然从库上查询的车子位置是在附近，但车子实际的位置可能已经移动到 1 公里外。这就给乘客带来了不好的体验，造成乘客呼叫了车后，车久久不来。

以上问题的解决方案有下面两个。

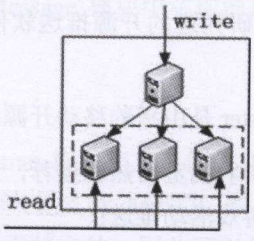
- MongoDB3.0 的 WiredTiger 存储引擎，该款存储引擎支持了文档级别的锁，但由于这款存储引擎不兼容 MongoDB 以前版本的数据，因此升级困难。
- MongoDB 的分片架构。对于大多数 LBS 后台来说，这是最理想的解决方案。

2. 分片架构

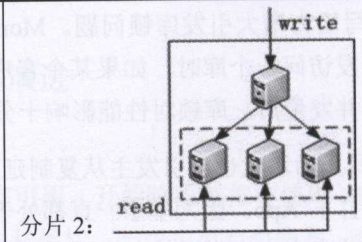
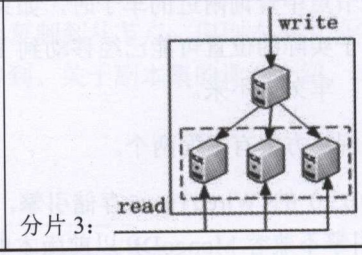
分片架构可以把集群中大量的数据读写请求分散到多个片上处理，每个分片是副本集的架构。关于副本集的详细介绍，请读者查看本书“8.4.3 分片”。

分片要注意合理的分片规则，通过合理的分片规则把 LBS 请求合理分散到各个分片。对于 LBS 应用，一个常用的分片规则是根据用户所在的地区分片，例如打车 App，一线城市的打车使用 LBS 的频率比二三线城市多，因此可以把所有用户的地理位置数据按照地区划分到不同的分片。由于 MongoDB 现在默认的分片策略还不能实现基于地理位置的分片，因此采用的是应用层分片方案。下面展示了一种分片方案，把所有 LBS 请求按照 3 个地区的划分在不同的分片，如表 9-6 所示。

表 9-6　把所有 LBS 请求按照 3 个地区分片

地区 1，包含北京、上海两个地区	 分片 1：

续表

地区 2，包含深圳、广州两个地区	
不包含在地区 1 和地区 2 的地区，都归为这个	

9.4 推送服务器后台架构

推送服务已经成了 App 必不可少的服务。架设推送服务除了可以使用第三方服务外，也有大量的开源技术可以选择。

现在推送服务主要分两块：Android 推送和 iOS 推送，笔者针对这两块分别讲述其架构。

9.4.1 Android 推送

Android 手机由于 Android 系统没有限制，当 App 进入了后台也能运行服务，所以 Android 可以基于长连接作推送，这就决定了 Android 的推送后台和一般基于 HTTP 的 App 后台是不一样的，从技术细节上，架构上也不一样。自主研发推送软件对一般的开发者来说有很大的难度，幸运的是现在有开源软件可以实现推送。

下面以笔者研究过的开源推送软件“Gopush-Cluster”为例，说明 Android 推送服务器的架构。

Gopush-Cluster 是由猎豹移动开源的推送软件，已经被广泛应用于猎豹移动下面的业务。

- 微看、猎豹浏览器热剧推荐，追剧提示；
- 游戏的游戏活动推送；
- 手机助手 App 推荐，广告；
- 电池医生实时推荐、求生手册的实时消息；

- PC 毒霸的指令下发（Web 换肤，升级提示）、漏洞泡泡等；
- PC 手机助手实时游戏、广告推荐；
- PC 猎豹浏览器上 WEB。

笔者和 Gopush-Cluster 主要开发者毛剑同学讨论技术的过程中，非常感谢毛剑同学不厌其烦的解答，帮助笔者解决了很多 Gopush-Cluster 设计上的疑惑。

关于 Gopush-Cluster 所使用的协议，可参考“9.1.2 协议”部分。

Gopush-Cluster 的架构如图 9-36 所示。

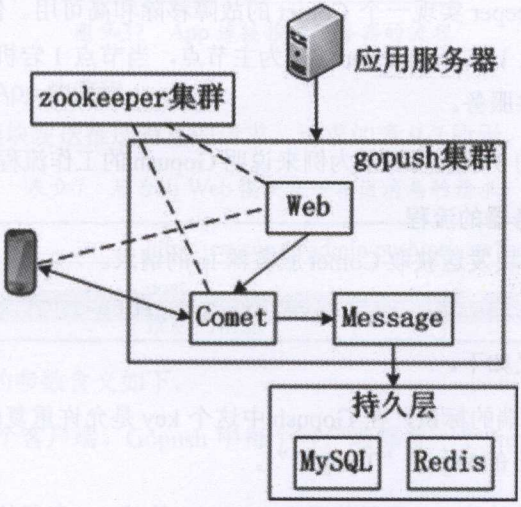


图 9-36　Gopush-Cluster 的架构图

主要分为四个模块来开发：Comet、Message、Web、third-part。

Comet

主要负责消息的推送，维护客户端连接。消息是有唯一 id 号，这个 id 号是依次自增的。当消息推送后需要在队列中排队，通过 PRC 传递到 Message 模块中实现消息的存储。这个模块是有状态的。

Message

主要负责消息的存取。当 Message 模块接收到 Comet 模块消息存储的 RPC（远程过程调用协议）请求后把消息持久化（目前开源版本的消息持久化支持 MySQL 和 Redis），同时接收来自 Web 模块的获取离线消息的请求。这个模块是无状态的，因此可以部署多个 Message 节点来应对 Comet 模块的请求和 Web 模块的请求压力。

Web

主要负责节点的询问（获取有效的 Comet 节点 IP 地址给客户端连接）、获取离线消息、后台节点的管理。节点询问是根据用户的 Key 使用一致性 hash 算法计算出该 Key 应该连接到哪个 Comet 节点，使用这种策略可以使海量用户均衡地分布到多个 Comet 节点，提高系统的并发能力。获取离线消息时，该模块发送 RPC 请求到 Message 节点，获取在持久化设备上的信息。这个模块是无状态的，可以用多个 Web 节点实现负载均衡和高可用。

thrid-part

系统还使用了 Zookeeper 实现一个 Comet 的故障移除和高可用。例如，Comet 节点 1 可以有一个备用节点 2，节点 1 在 Zookeeper 注册为主节点，当节点 1 宕机后，Zookeeper 会选举节点 2 为可用的节点来提供服务。

下面以推送服务中的 3 个常用场景为例来说明 Gopush 的工作流程。

1. App 连接推送服务器的流程

（1）App 向 Web 模块发送获取 Comet 服务器 ip 的请求。

```
http://test.com/1/server/get?key=Terry22&proto=1
```

上面各个参数的含义如下。

key：每个 App 客户端的标识，在 Gopush 中这个 key 是允许重复的，因此可实现同账号的多设备登录。这里该 App 的 key 是“Terry22”。

proto：通信所使用的协议，1 表示使用 WebSocket 协议，2 表示使用 TCP 协议。

（2）Web 模块从 Zookeeper 中获取有效的 Comet 模块信息，并根据一致性 hash 算法（一致性 hash 算法的描述读者请查看本书“9.2.4 缓存架构的演进”这节中的描述）得到该 Key 应该连接的 Comet 模块，返回下面的 JSON 数据给 App。

```
{
    "ret": 0,
    "data": {
     "server": "183.54.1.1:8065"  //IP:Port
    }
}
```

（3）App 从返回的 JSON 数据中得到 Comet 模块的 IP，使用 Comet 模块的协议格式连接到 Comet 模块，这样 App 就成功连接了推送服务器。

App 连接推送服务器的流程如图 9-37 所示。

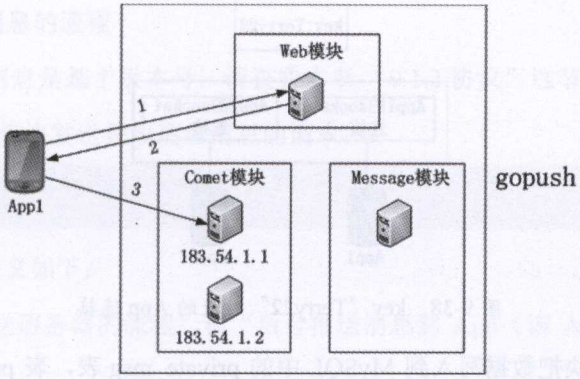


图 9-37　App 连接推送服务器的流程

2. 后台推送消息到 App 的流程

（1）后台向 Web 模块发送推送消息的请求，请求如表 9-7 所示。

表 9-7　后台向 Web 模块发送推送消息的请求

HTTP URL	http://test.com/1/admin/push/private?key=Terry22&expire=100000
HTTP method	post
HTTP body	推送的消息

其中 HTTP URL 中的参数含义如下。

key：标识发送到哪个客户端。Gopush 中每个客户端都有一个 key 做标识，这个 key 是允许重复的。

expire：消息过期时间，单位：秒(s)。App 获取离线消息时，如果发现该消息存储超过一段时间就不再推送给 App，这段时间的长度由这个 expire 参数决定。

（2）Web 模块根据一致性 hash 算法得到该 Key 连接的 Comet 模块（在集群中可能有多个 Comet 模块，需要通过一致性 hash 算法得到该 Key 所在的 Comet 模块，关于“一致性 hash 算法”，请读者查看本书“9.2.4 缓存架构的演化”这节），Web 模块通过 RPC 调用 Comet 模块中的接口，把 Key、expire、推送的消息这些参数传递到 Comet 模块。

（3）Comet 模块收到参数后，检查是否有属于该 Key 的 App 连接上 Comet 模块（Comet 模块内部采用 hashmap 的方式，一个 Key 可以对应 1 个或多个 App 的 Socket 连接，如图 9-38 所示），把消息推送到对应的 App 上，该消息上附带一个版本号。

（4）Comet 模块通过 RPC 把 Key、expire、推送的消息、消息的版本号传递到 Message 模块保存。目前 Gopush 只支持 MySQL 和 Redis 两种存储方案，开发者可以通过实现相应的存储接口添加新的存储方案。

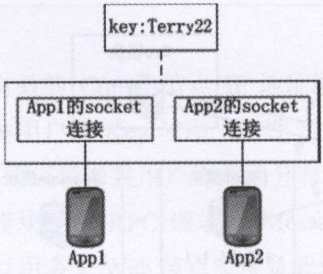


图 9-38　key“Terry22”对应的 App 连接

（5）Message 模块把数据写入到 MySQL 中的 private_msg 表，表 private_msg 各个字段的含义如表 9-8 所示。

表 9-8　表 private_msg 各个字段的含义

字段	含义
skey	Key
mid	版本号
ttl	expire（过期时间）
msg	消息
ctime	创建时间
mtime	修改时间

后台推送消息到 App 的流程如图 9-38 所示。

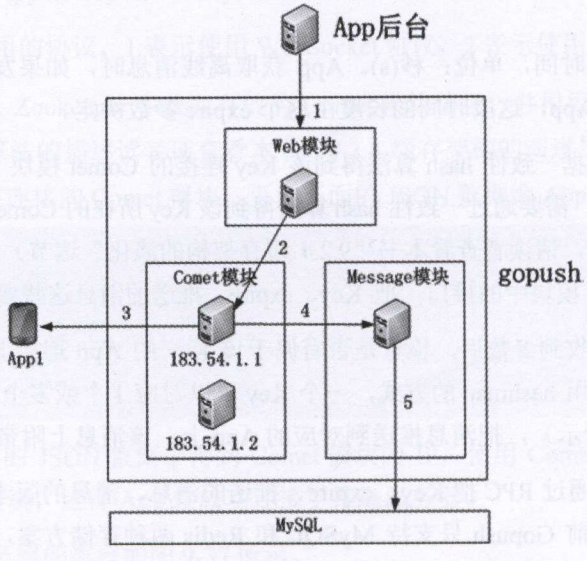


图 9-38　后台推送消息到 App 的流程

3. App 获取离线消息的流程

Gopush 获取离线消息是基于版本号，请查看本书“9.1.2 协议”这节。

（1）App 向 Web 模块发送获取离线消息的请求：

```
http://test.com/1/msg/get?k=Terry22&m=13999084541836408 获取该 key 的所有离线消息
```

上面各个参数的含义如下：

k：“App 连接推送服务器的流程”和“后台推送消息到 App（该 App 在线）的流程”中的 Key 含义相同。

m：最后接收的消息版本号。

（2）Web 模块通过 RPC 调用 Message 模块的接口获取离线消息。

（3）如果离线消息是存储在 MySQL，Message 模块通过下面的 SQL 语句获取相应的 Key。

```
SELECT mid, ttl, msg FROM private_msg WHERE skey=k AND mid>m ORDER BY mid
```

该 SQL 语句获取所有属于该 Key 的数据，而且数据的版本号必须大于传入的 m 值。

（4）Message 模块检查所获取的 MySQL 的数据的 ttl 值（过期时间），如果发现该条数据已过期，则不返回给 Web 模块。

（5）Message 模块把合适的数据返回给 Web 模块。

（6）Web 模块把 Message 模块返回的离线消息传给 App。

App 获取离线消息的流程如图 9-39 所示。

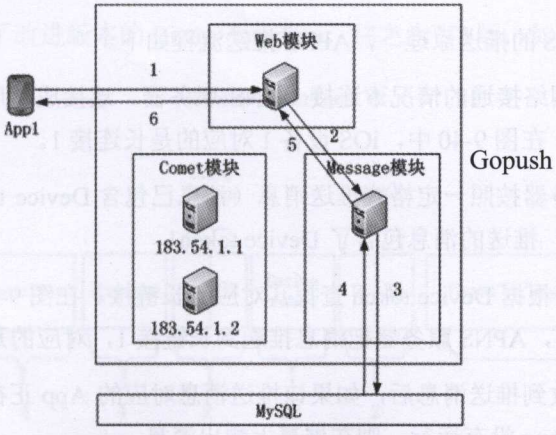


图 9-39　App 获取离线消息的流程

9.4.2 iOS 推送

iOS 推送必须用到 APNS。APNS 是 Apple Push Notification Service（Apple Push 服务），由于 iOS 系统的限制，应用是不允许在后台运行并连接网络的。如果 App 在 iOS 上没有运行，但是开发者却想推送消息给使用该 App 的 iOS 用户，只能通过 APNS 推送消息。

1. APNS 原理

APNS 推送有一个重要概念：Device token。Device token 是 iOS 设备上某个 App 在 APNS 上的唯一标识符，通过 Device token，APNS 就知道把消息推送到哪台 iOS 设备上由哪个 App 接收。

APNS 的推送原理如图 9-40 所示。

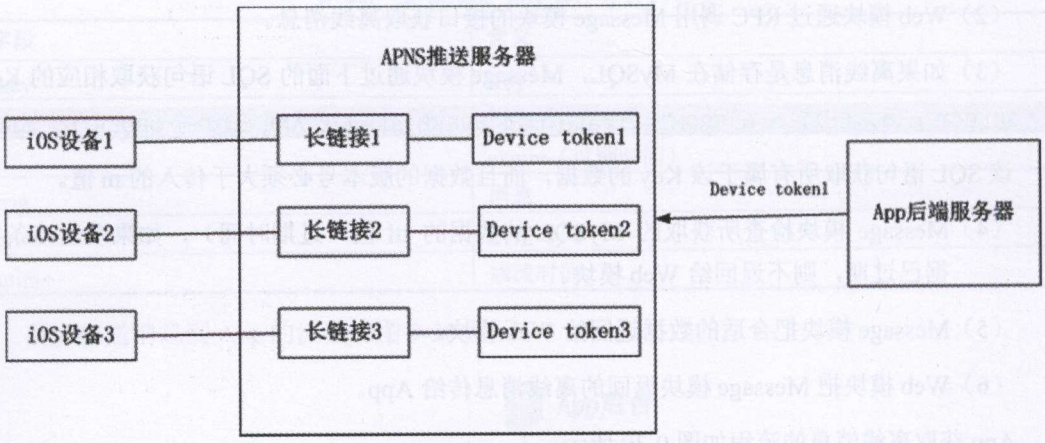


图 9-40 APNS 的推送原理

根据图 9-40“APNS 的推送原理”，APNS 推送流程如下。

（1）iOS 设备在网络接通的情况下连接 APNS 服务器，连接成功后和 APNS 服务器保持一个长连接。在图 9-40 中，iOS 设备 1 对应的是长连接 1。

（2）App 后台服务器按照一定格式推送消息（消息已包含 Device token）到 APNS 服务器。在图 9-40 中，推送的消息包含了 Device token1。

（3）APNS 服务器根据 Device token 查找其对应的长链接。在图 9-40，Device token1 对应的是长链接 1，APNS 服务器把消息推送到长链接 1，对应的是 iOS 设备 1。

（4）iOS 设备 1 收到推送消息后，如果该推送消息对应的 App 正在运行，则通知 App 处理，如果该 App 没有运行，则在屏幕上弹出消息。

2. APNS 推送协议分析

APNS 上的推送接口称为 Binary Interface，App 后台连接 APNS 服务器后必须按照 Binary Interface 格式推送消息。

如图 9-41 所示是最初版本的 Binary Interface，称之为 V1 版。

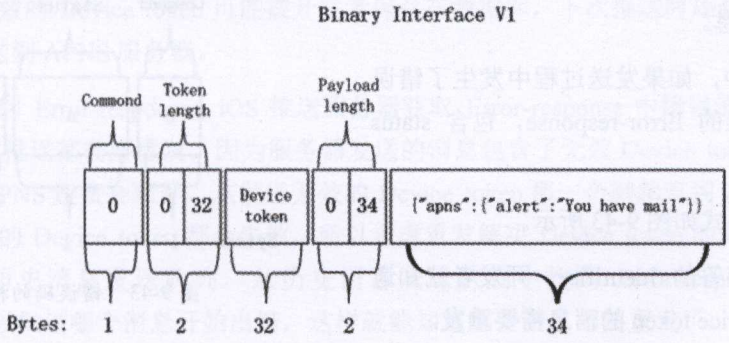


图 9-41　最初版本的 Binary Interface

这个版本的接口有如下两个问题。

- 没法确定消息是否发送成功。
- 当发送了一个无效 Device token 后，APNS 和 iOS 设备的连接会在短时间内断开，在连接断开之前发送的 Device token 也会失效。这就是说，当连续推送一批消息（里面可能包含无效的 Device token）后如果链接断开，没法确定推送给哪些 Device token 的消息是已经发送成功，推送给哪些 Device token 的消息需要重发，这就造成了用户经常收不到推送的情况。

接下来苹果推出了改进版本的 Binary Interface，称之为 V2 版，如图 9-42 所示。

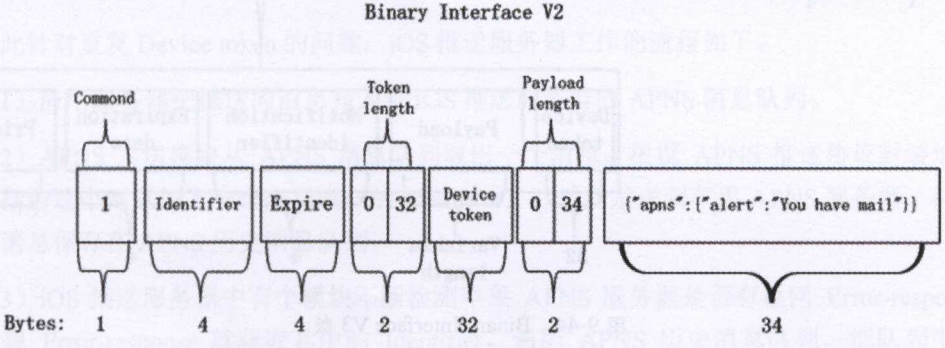


图 9-42　改进版本的 Binary Interface

读者可看到 V2 版的协议增加了如下两个字段。

- Identifier：用于识别一条消息，可以是任意值。如果发送出现问题，在 APNS 返回的错误应答中会包含出问题的 Identifier。
- Expiry：离线消息超时的时间。如果为 0 或者小于 0，APNS 服务器不会保存这条消息。

在 V2 版中，如果发送过程中发生了错误，会返回一个错误的 Error-response，包含 status 和 Identifier。

错误码的格式如图 9-43 所示。

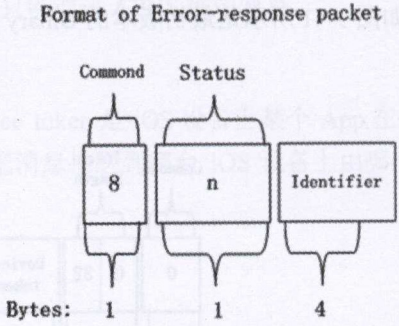


图 9-43　错误码的格式

根据错误应答的 Identifier，开发者就知道推送给哪些 Device token 的消息需要重发。

Binary Interface V2 有个不便的地方：每个推送只能给一个 Device token 推送消息，如果要给上百万台设备推送消息那效率非常低，于是苹果推出了 Binary Interface V3 版，如图 9-44 所示。

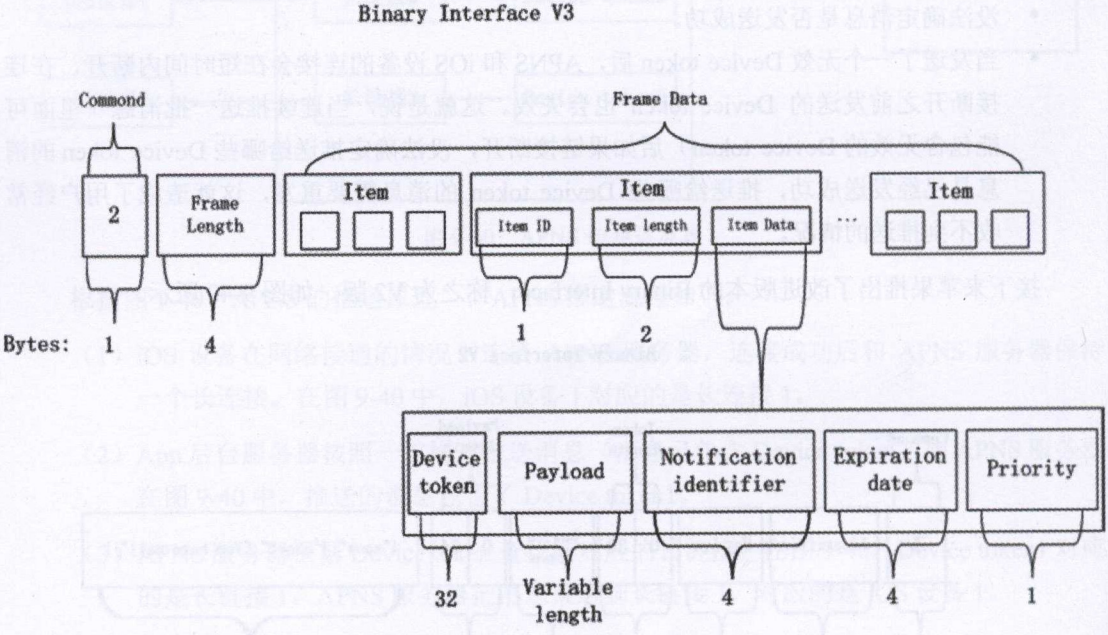


图 9-44　Binary Interface V3 版

从图 9-44 中可看到，在 Binary Interface V3 版中一次给多个 Device token 发送消息。

3. iOS 推送服务器架构

服务端推送 APNS 过程中需要注意推送了无效 Device token 的问题。

为什么会有无效的 Device token 呢？因为当用户卸载了 App 后，用户的 Device token 就失效，但是这个失效的 Device token 可能被开发者保存在数据库，下次推送时还会把这个失效的 Device token 推送到 APNS 服务器。

如果推送返回 Error-response，iOS 推送服务器获取 Error-response 中错误消息的 Identifier，该 Identifier 后的推送都需要重发。因为服务器发送的消息包含了无效 Device token 后隔一段时间，服务器和 APNS 连接会断开，从发送无效的 Device token 那一个刻起直到 APNS 连接断开，这段时间所发送的 Device token 都会失效。所以考虑重发解决 Device token 的问题时可以把发送消息保存到历史消息发送队列，从历史消息发送队列中找到 Error-response 中返回的 Identifier，就可以知道哪个消息开始出错，这样就能知道哪些消息需要重发了。

APNS 历史消息队列如图 9-45 所示。

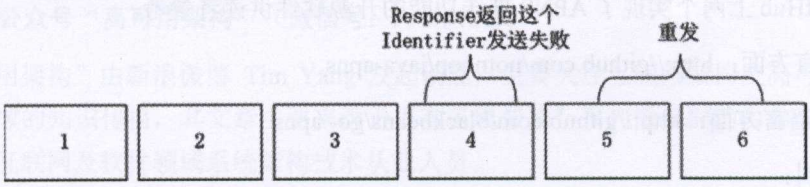


图 9-45　APNS 历史消息队列

假设当 Error-response 中返回的 Identifier 为 4，那么后面 Identifier 的 5 和 6 都需要重发。

另外 APNS 的 feedback service 会返回那些已经卸载的设备的 Device token，定期在数据库中删除这些无效的 Device token 以提高推送的成功率。

因此针对重发 Device token 的问题，iOS 推送服务器工作的流程如下。

（1）应用服务器把推送的消息写入到 iOS 推送服务器的 APNS 消息队列。

（2）APNS 发送模块从 APNS 消息队列取出一个消息，根据 APNS 推送协议封装这个消息（包括给这个消息加上一个全局唯一的 Identifier），然后发送到苹果 APNS 服务器，同时把发送的消息保存在 APNS 历史消息队列。

（3）iOS 推送服务器中有个模块不断检测苹果 APNS 服务器是否有返回 Error-response，一旦发现 Error-response 就获取其中的 Identifier，遍历 APNS 历史消息队列，把队列中这个 Identifier 后面的所有消息重新放入到 APNS 消息队列，让 APNS 发送模块重新发送这些消息。

上面所描述的 iOS 推送服务器的工作流程如图 9-46 所示。

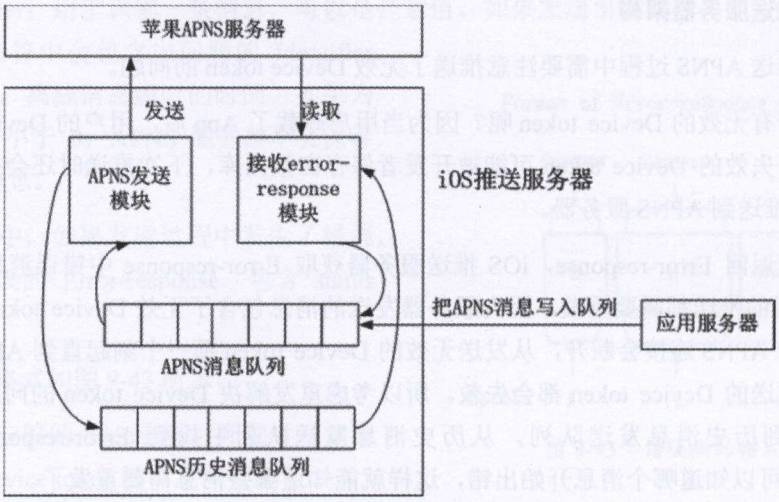


图 9-46　iOS 推送服务器的工作流程

附带 GitHub 上两个实现了 APNS 推送功能的开源软件供读者参考。

Java 语言方面：http://github.com/notnoop/java-apns

Golang 语言方面：http://github.com/blackbeans/go- apns

参考资料：

https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Appendixes/BinaryProviderAPI.html

9.5　获得更多 App 后台架构资料

下面列举了笔者学习 App 后台架构知识的 5 个网站。

1. InfoQ（http://www.infoq.com/cn）

InfoQ 是为了促进软件开发领域知识与创新的传播而创立的，其提供了新闻、文章、视频演进和采访等资讯服务。

在 InfoQ 中文站点上（http://www.infoq.com/cn），能看到国内移动互联网中一线技术负责人的大量技术演讲、最新的技术新闻、介绍前沿技术的文章等资讯。

InfoQ 还举办 QCon 全球软件开发大会（InternationalSoftware Development Conference）和全球架构师峰会（International ArchitectSummit，以下简称 Archsummit），在这两个大会上能了解到业界最新的前沿技术，还有和大量同行沟通的机会。

2. 七牛开发者最佳实践日（http://best.qiniu.com/archives/category/developers-best-practice）

“开发者最佳实践日”是由七牛云存储发起并联合各个技术高手为开发者举办的系列技术沙龙，关注开发者在实际中可能遇到的各种技术问题。致力于为勇于创新的开发者们提供行业内最前沿、最热门的技术，以技术驱动应用创新，让更多的开发者享受技术带来的生活乐趣。

3. UPYUN Open Talk（http://opentalk.upyun.com/show/issues）

Open Talk 是由 UPYUN 发起的系列主题分享沙龙，为互联网从业人员呈现以技术为主，同时涵盖产品、营销、融资、创业经验等各个方面的专业知识。

笔者所在公司的 CEO 何少岳博士在 9 月份广州的 Open Talk 上进行了分享，主题为《技术人如何转型创业》，有兴趣的读者可以去看一下该演进视频。

4. 微信公众号“高可用架构”（微信号：ArchNotes）

“高可用架构”由新浪微博 Tim Yang 发起创建，主要关注互联网架构及高可用、可扩展及高性能领域的知识传播，其文章主要来源于“高可用架构”系列微信群的内容分享。订阅用户覆盖主流互联网及软件领域系统架构技术从业人员。

高可用架构系列微信群是一个社区组织，其精神是“分享+交流”，提倡社区的人人参与，同时从社区获得高质量的内容，该系列群由近 2000 位主要来自国内外互联网行业的架构师、首席架构师、CTO、资深工程师等组成，包括大量 QCon 明星讲师及互联网资深讲师参与分享优质内容。下面是该公众号最近的分享列表，里面有大量的实用知识，部分文章如下。

- 《支撑微博千亿调用的轻量级 RPC 框架：Motan》
- 《高德架构迁移阿里云的实践：从 7 个私有机房到公有云的历程》
- 《对话架构师：亿级短视频社交美拍架构实战》
- 《来自 Google 的高可用架构理念与实践》
- 《如何设计类似微信的多终端数据同步协议 | Grouk 实践分享》
- 《单表 60 亿记录等大数据场景的 MySQL 优化和运维之道 | 高可用架构》

高可用架构群上的精品文章也会由电子工业出版社以书籍的形式出版，敬请期待。

5. 新浪微博 @微博平台架构

微博平台架构官方微博，不定期分享微博的技术架构。今年微博把新兵训练营（新兵训练营主要是面向应届毕业生，举办的目的在于让新同学系统化并且有针对性地了解微博平台的核心技术及核心业务）的培训课程整理成一系列的文章发表在该微博，使众多不在微博平台工作的同学也可以对微博平台的核心技术及核心业务有所认识，文章列表如下。

- 《平台服务部署及 Web 框架》
- 《平台 RPC 框架》
- 《编写优雅的代码》
- 《环境及工具》
- 《分布式缓存》
- 《海量数据存储基础》
- 《一次服务上线》
- 《Feed 架构介绍》
- 《Unread 架构介绍》

第 10 章
App 后台架构的演进

在本书前面的章节中，按照 App 后台架构所需知识的层次，分别介绍了下面 3 部分的内容.

- App 后台架构中所用的技术讲解。
- App 后台架构中常用软件的运维知识。
- 4 类 App 后台架构剖析。

在本章中将会融合前面 3 部分的知识，结合笔者参与过的 App 项目的后台架构经验，讲述架构的核心要素、架构选型的特点，以及 App 后台架构的演进。

10.1 架构的核心要素

本章谈论的是架构，需要先弄明白：什么是架构？

百科中关于软件架构的定义如下：

软件架构是有关软件整体结构与组件的抽象描述，用于指导大型软件系统各个方面的设计。

把软件架构的定义对应“App 后台架构”，笔者的定义是：由 App 后台各个组件的功能描述、相互关系构成的整体系统。

在 App 后台的架构中除了考虑 App 后台的功能需求外，还需要考量下面 5 个核心的要素，如图 10-1 所示。

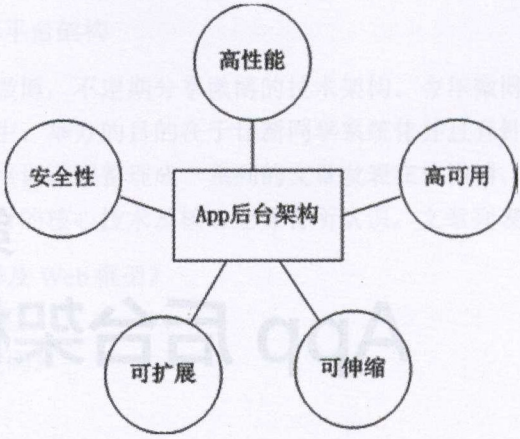


图 10-1　App 后台架构的 5 个核心要素

App 后台架构的 5 个核心要素如下。

- 高性能
- 高可用
- 可伸缩
- 可扩展
- 安全性

下面逐一讲述 App 后台架构的 5 个核心要素。

10.1.1　高性能

高性能是 App 后台的一个重要指标，除非其 App 是独一无二、不可或缺的，否则任何一个用户都无法忍受一个响应速度极慢的 App，特别是移动互联网这个讲求快速响应的环境。当 App 性能表现不理想、体验差时，用户就自然而然地放弃使用这个 App，甚至会投入到竞争对手的怀抱。

性能问题也是驱动架构发展的最直接力量，因为性能问题是最容易被用户感知的，当打开一个 App 的界面后，以前在 1 秒之内就能从 App 后台获取数据并展示数据完毕了，但用户量增大后，打开同样的界面需要几秒甚至几十秒，那么开发者就要考虑改进架构。

从 App 发出请求到 App 后台返回响应结果，这一过程如图 10-2 所示。

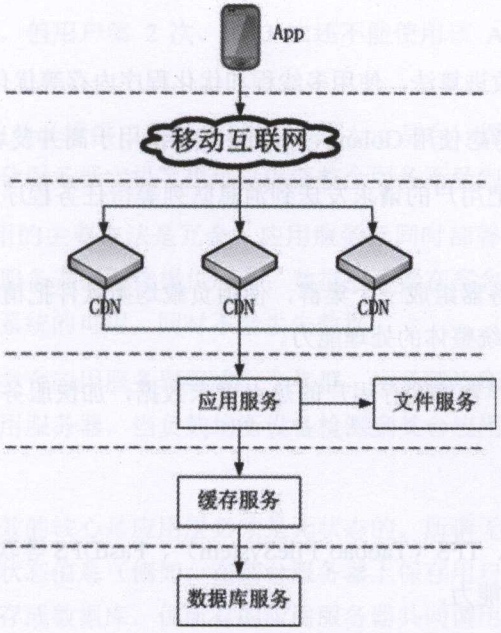


图 10-2　App 发出请求到 App 后台返回响应结果的过程

在 App 发出请求到 App 后台返回响应结果时，按照图 10-2 中的展示，每层可以有下面的措施以提高系统的性能。

1. App 层

（1）图片、音频、视频等静态资源，第一次下载后可以缓存在手机的 SD 卡，这样就不用每次显示图片都需要下载。

（2）对于 Feed、通知等内容，使用增量更新的技术，减少服务器的负担和使用的流量。关于增量更新的详细讲解可查看本书“3.5.2 数据增量更新策略”一节。

（3）根据 App 当前的网络环境下载不同的图片数据。例如，使用查看原图的功能，如果是在 Wi-Fi 环境下就下载原图，如果是 3G 等移动网络下就下载分辨率比原图低但也清晰可见的缩略图。

2. 网络传输层

（1）使用 CDN 技术，让用户在最近的机房下载图片、音频、视频等静态资源，减少网络传输的时间，使用户获得更快的下载速度。

（2）在应用服务器部署反向代理服务器、缓存热点文件，使请求在到达应用服务器前返回静态资源，减轻应用服务器的负担，减少请求的时间。

3. 应用服务层

（1）在代码层面，改进算法，使用多线程和优化程序内存等优化方法。

（2）在语言层面，考虑使用 Golang、Erlang 等更适用于高并发场景的语言。

（3）通过异步操作把用户的请求发送到消息队列等待任务程序处理，减少请求的等待时间。

（4）将多台应用服务器组成一个集群，使用负载均衡软件把请求按一定的规则分发到每个应用服务器上，提高系统整体的处理能力。

（5）使用分布式缓存软件缓存用户的热点请求数据，加快服务器的响应时间，减轻数据库的负担。

4. 文件服务层

（1）使用 MogileFS、TFS（Taobao FileSystem）、FastDFS 等软件架设一个分布式文件系统，提供整体的文件处理能力。

（3）使用七牛、又拍、UCloud 的对象存储（UFile）等第三方文件云存储服务，把文件存放在云服务器上，从而在架构上去掉文件服务器，笔者推荐这种做法。

5. 缓存层

可使用 Redis、Memcached 或者云服务器的云缓存服务，这些基于内存的缓存服务已经提供了足够高的性能。

6. 数据库层

（1）数据库前加一层或多层缓存，挡住大部分的热点请求，使大部分的请求不穿透到数据库，减轻数据的压力。

（2）对于 MySQL 数据库，可以使用读写分离、分表、分库等成熟的技术；对于 NoSQL 类型的产品，例如 MongoDB，可以使用其原生的副本集、分片等机制，提升其性能。

（3）使用 Facebook 开源的 FlashCache 技术，把传统硬盘上的热数据缓存在 SSD 硬盘上，冷门数据保存在传统硬盘上，利用 SSD 优秀的读性能增加。

10.1.2 高可用

对于很多著名的 App 来说，高可用是其架构中一个很重要的要素。

在移动互联网年代，用户对 App 的容忍性是极其有限的，当用户第 1 次不能使用这个 App

的服务时，可能就忍一下，但用户第 2 次、第 3 次还不能使用该 App，那用户可能就把这个 App 从手机上卸载了。

高可用就是要保证为 App 提供 7×24 小时服务的 App 后台，服务器不能随便宕机，或者在一个服务器集群中，部分服务器宕机了也可以保证整个服务不受到影响。

保证 App 后台高可用的主要方法是冗余：应用服务器同时部署两台以上，保证其中一台服务器出问题了另外一台服务器能继续提供服务；数据服务器在多台服务器上相互备份，任何一台服务器宕机也能保证系统的可用，同时不会丢失数据。

对应用服务器而言，多台应用服务器组成一个集群，由负载均衡设备把请求按照一定的策略分发到集群中的每个应用服务器，当负载均衡设备检测到某台应用服务器宕机，就把该台应用服务器从集群中移除。

保证负载均衡策略有效的核心是应用层必须是无状态的。所谓无状态，是指任意一台应用服务器上不会保存用户的状态信息（例如，在某台服务器上保存用户已经登录的凭证）。用户的状态信息可以存储在缓存或数据库，供所有的应用服务器共同调用。当应用层是无状态的，那么通过负载均衡设备把请求分发到任意一台应用服务器，每台服务器的处理都没差别。当应用层是有状态的，例如把用户的登录信息保存到其中一台应用服务器，当负载均衡设备把请求分发到其他应用服务器时，该用户的登录信息就会丢失。

应用服务器的高可用原理如图 10-3 所示。

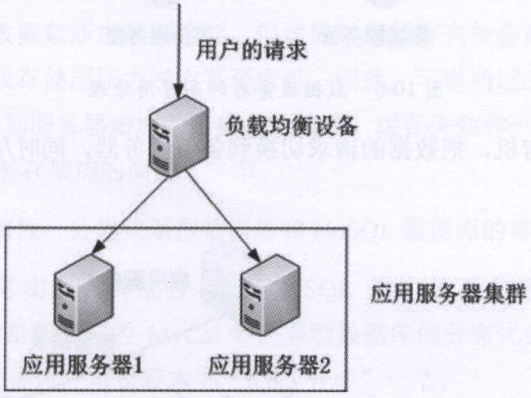


图 10-3　应用服务器的高可用原理

当负载均衡设备检测应用服务器 2 宕机了，把应用服务器 2 从集群中移除，只把请求分发到应用服务器 1，如图 10-4 所示。

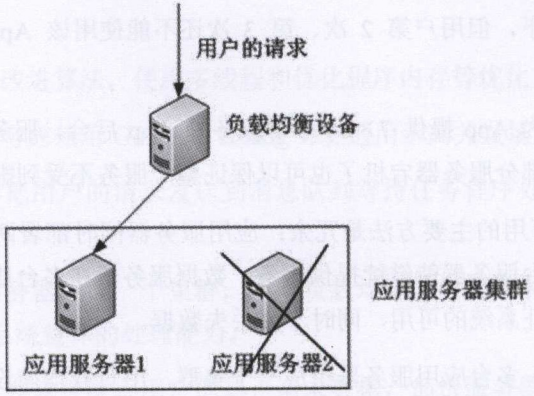


图 10-4　应用服务器 2 宕机后的负载均衡设备的分发图

对于数据服务器（包含数据库、缓存、文件）而言，保证高可用需要对存储的数据进行实时备份，当数据服务器宕机后，立刻把数据的读写请求切换到备份服务器上，同时尽快修复宕机的服务器，以保障冗余。数据服务器的高可用原理如图 10-5 所示。

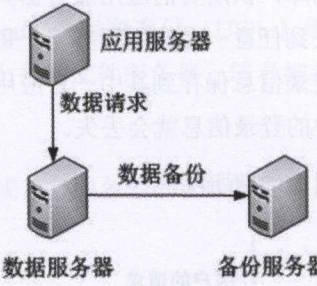


图 10-5　数据服务器的高可用原理

检测到数据服务器宕机，把数据的请求切换到备份服务器，同时尽快修复宕机的服务器，如图 10-6 所示。

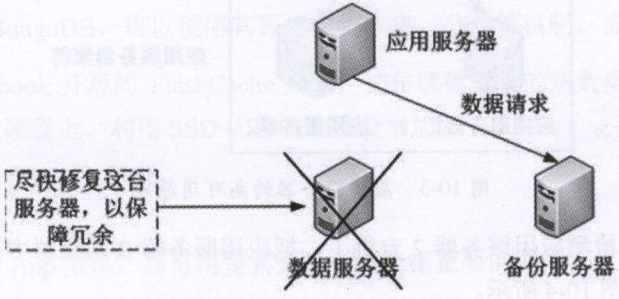


图 10-6　数据服务器宕机时的处理图

10.1.3　可伸缩

大型的 App 后台在面对海量用户的高并发访问量和海量数据的存储需求时，无论配置多么高级，一台服务器到了一定的阶段后总会没法满足需要，这时就要考虑使用多台服务器提供服务。可伸缩是指通过往集群中添加机器，应付不断增大的访问压力和数据存储需求。

对于应用服务器而言，可伸缩性是指当访问量增大后，往集群中添加服务器。用户通过负载均衡设备分发请求到集群中的服务器，使每台服务器分担集群中不断增大的访问量。应用层实现可伸缩的前提是应用服务器是无状态的，即应用服务器不保存用户的信息，用户请求分发到任意一台服务器都是对等的。

数据存储的可伸缩性分为文件数据的可伸缩性、缓存数据的可伸缩性、数据库的可伸缩性。

文件数据的可伸缩性，使用分布式文件存储软件很容易实现。以分布式文件存储软件 Fastdfs 为例，当发现集群中的文件容量快到瓶颈时，有下面两个方法可以增加集群的文件容量。

- 在某台服务器上添加硬盘，就能给集群扩容。如果使用的是 UCloud 等云服务器，其提供了给云服务器添加云硬盘的服务，通过简单的几步操作就能把一个新硬盘挂载到云服务器，无须停机，保证服务不停止。
- 在集群中添加服务器，Fastdfs 内部检测到新的服务器后，会把文件保存在新的服务器上。

缓存数据实现可伸缩，由于添加了新的服务器会导致之前的路由失效，造成大多数缓存都没法命中。虽然可以把数据重新加载到缓存，但这段时间缓存失效会造成数据请求的压力直接达到相关的存储层，造成存储层压力过大甚至宕机。因此，可以通过改进的路由算法（例如一致性 Hash 算法）减少添加服务器造成的路由失效情况。缓存失效和一致性 Hash 算法的详细描述，可查看本书“9.2.4 缓存架构的演进”一节。

关于数据库的可伸缩性，分为关系型数据库和 NoSQL 数据库的可扩展性。

MySQL 使用分库可实现可伸缩性，但 MySQL 原生是不支持分库的。MySQL 通过 Amoeba 和 Cobar，或者是新兴起的 MyCat 等关系型数据库的分布式处理系统，就能实现分库。关于 MySQL 分库的详细描述，可查看本书“6.6.3 分库”一节。

NoSQL 数据库天生就是为了分布式存储设计的，很简单就能实现伸缩性。例如，NoSQL 数据库 MongoDB 使用分片策略就能实现伸缩性。关于 MongoDB 分片功能的详细描述，可查看本书“8.4.3 分片”一节。

10.1.4 可扩展

在移动互联网时代，App 的一个重要特点是迭代的速度非常快，可能新的产品出来后，连用户从界面上都看不出是以前的那个 App。

App 快速迭代的特点注定了需求是多变的，可扩展性就是关注在需求多变的情况下怎样对现在的架构影响最少。

可扩展性的核心是减少模块间的耦合度，每个模块都尽量少依赖其他模块，这样其中一个模块的变化对其他模块的影响减少。实现可扩展性有下面的 3 种方式。

- 消息队列：生产者（某个业务模块）将消息放到消息队列，消费者（另外的业务模块）将消息从消息队列中取出来进行处理。通过动态增减消息，再加上消息的生产者和消费者分离的方法，就能降低模块间的耦合程度。
- 分布式服务：把业务中可复用的模块抽离成一个独立的服务，对其他模块提供可复用的服务，通过分布式服务框架供其他模块调用。新增的业务通过调用可复用的服务实现其需要的业务逻辑，减少了开发量。当可复用的服务需要改变其业务逻辑时，由于其他模块都是调用同一个可复用服务，可使代码的修改量减到最少。
- 开放式 API：从商业的角度来说，把自身的业务封装成开放式 API 供其他开发者调用，也是实现系统可扩展性的一个重要方法。国外的 Facebook、Twitter，国内的淘宝、腾讯、微博等企业，大量的开发者基于其开放式 API 创建了海量应用，极大丰富了产品平台。

10.1.5 安全性

安全性就是保证用户的核心数据不被非法人员盗窃。

国内外发生的多宗用户的核心数据泄露事件，给相关的 App 带来了极坏的影响，导致大量用户流失。

对于和金钱交易相关的 App，安全性更为重要，如果因为安全问题造成用户的金钱损失，轻则用户流失，严重则会引起法律纠纷。

10.2 架构选型的要点

开发 App 后台，当需求初步确定后，就要考虑用什么技术去实现架构。基于架构设计的递进关系，笔者认为架构选型有如下两个要点。

- 用成熟稳定的开源软件。
- 优先选择云服务。

10.2.1　用成熟稳定的开源软件

用开源软件方案有下面两个原因。

- 节省资金。
- 非开源方案，如果不是被广泛使用而且没提供技术支持的话，出了问题也很难从代码层面查找。

以前由于开源软件的不成熟，没法满足公司业务上的需求，某些大型公司的基础组件需要采用自研的方案。随着时间的发展，众多开发者（甚至是公司）参与到开源软件的编写与反馈迭代工作，开源软件越来越成熟，众多开发者对其也越来越熟悉，使用开源软件在短时间内就能搭建一个成熟稳定的 App 后台。

当然也有不少的公司在开源软件的基础上，结合自身的业务特点对开源软件进行改造，例如，淘宝就把其定制开发的 Nginx 版本命名为 Tengine，并对其开源。

笔者之所以一直强调用成熟稳定的开源软件，是基于以下几点理由。

- 由于成熟稳定的开源软件用户基数足够大，其内部出现的各种代码错误和功能缺陷极大可能已经被其他开发者反馈给开源软件团队修复。
- 在使用开源软件的过程中出现的各种在使用者认知水平之外的问题，由于开源软件用户基数足够大，可能别的开发者也曾经遇过并解决了这些问题，能通过搜索引擎在网络上找到合适的解决方案。当使用非成熟稳定的开源软件时，出了什么问题在搜索无效的情况下，只能查看源码来定位问题，或者和软件的开发者沟通，看看有哪些解决问题的方案。但一般软件的开发者只留下 E-Mail 等非即时沟通方式，沟通并解决一个问题所需的时间周期很漫长。
- 由于成熟稳定的开源软件被广大开发者所熟悉，也有庞大的用户群。在移动互联网行业人员的流动性比较大，如果采用自研的软件方案，新招聘的人员需要花大量的时间熟悉自研的软件方案，而采用成熟稳定的开源软件就不存在这个成本。

下面以这个具备了基本社交功能的 App 为例，列举其功能和项目管理中可供选择的开源软件方案。

- 使用手机号、微博、QQ 注册和登录。
- 手机号注册需要使用验证码。
- 用户之间能互相添加为好友。

- 能搜索 App 内的用户。
- 能查看附近的人。
- 用户能发表类似微博的东西（包括文字、图片、声音和地理位置），并在 App 首页显示所有好友发表的内容。
- 用户可以把内容分享到社交网络。
- 聊天功能，能发送文字、图片、声音和地理位置。
- 运营方可以给 App 用户推送各种产品消息、最新的通知等。

从上面这个 App 的功能实现和项目管理中可供选择的开源软件方案如表 10-1 所示。

表 10-1　可供选择的开源软件方案

功能	可使用的开源软件
项目管理工具	Mantis、BugFree
代码管理工具	SVN、GIT
编程语言	PHP、Java、Python 等
服务器软件	CentOS、Ubuntu
HTTP 服务	Nginx、Tomcat、Apache
负载均衡	Nginx、LVS、HAProxy
邮件服务	Postfix、Sendmail
消息队列	RabbitMQ、ZeroMQ、Redis
文件系统	Fastdfs、mogileFS、TFS（Taobao FileSystem）
Android 推送	Androidpn、gopush
iOS 推送	Javapns、Pyapns
LBS	MongoDB
聊天	Openfire、ejobberd
监控	ngiOS、zabbix
缓存	Memcache、Redis
关系型数据库	MySQL、postgreSQL
NoSQL 数据库	Redis、MongoDB、Cassandra
搜索	Coreseek、Solr、ElasticSearch
图片处理	GraphicsMagick、ImageMagick
分布式访问服务	dubbo、dubbox

10.2.2　尽可能使用云服务

移动互联网时代涌现了大量小而美的创业型公司，这些公司由于受到资金的限制，研发人员的配置非常精简，典型的研发人员配置如下。

- Android 开发 1~2 人。
- iOS 开发 1~2 人。
- App 后台开发 1~2 人。

在前面章节“10.2.1 用成熟稳定的开源软件方案”一节中以一个基本的社交 App 为例，用表列举了开发这个 App 所需要使用的开源软件，里面所需要使用的开源软件多达 20 个，想想都觉得可怕，一个开发人员掌握 20 个开源软件需要付出多少精力！这里的掌握还是指基本的配置和使用，如果需要对其原理和运作机制有深入的理解，所付出的精力更是无法评估。

随着云平台的不断发展，越来越多可复用场景的需求以云的形式提供服务，云服务也越来越完善。云服务和使用复杂多样的开源软件进行配置、运维相比，能极大地减少开发人员认知成本和运维成本，从而把精力专注于业务。

因此，笔者推崇创业公司的 App 后台的架构原则是，**“尽量使用成熟可靠的云服务和开源软件，自身只专注于业务逻辑”**。

对应 10.2 小节的 App，其功能实现和项目管理中可供选择的云服务如表 10-2 所示。

表 10-2　可供选择的云服务

功能	可使用的云服务
项目管理工具	Teambition、Tower
代码管理工具	GitHub
负载均衡	阿里云负载均衡服务 SLB， UCloud 负载均衡服务 ULB
邮件服务	SendCloud、MailGun
消息队列	阿里云消息通知服务 MNS
文件系统	七牛、又拍云， 阿里云对象存储 OSS， UCloud 对象存储 UFile
Android 推送	极光、个推、百度推送
iOS 推送	极光、个推、百度推送
聊天	融云、环信
监控	监控宝、云服务器自带的监控服务
缓存	阿里云开放缓存服务 OCS， UCloud 云内存存储 UMem
关系型数据库	阿里云云数据库 RDS， UCloud 云数据库 UDB

续表

功能	可使用的云服务
NoSQL 数据库	阿里云开放结构化数据服务 OTS， UCloud 云内存存储 UMem
搜索	阿里云开放搜索服务 OpenSearch
图片处理	七牛、又拍云， 阿里云对象存储 OSS， UCloud 对象存储 Ufile 等
分布式访问服务	阿里云企业级分布式应用服务 EDAS
防火墙	阿里云云盾， UCloud 防火墙
短信发送	bmob、shareSDK、Luosimao
社交登录分享	shareSDK

由于现在云服务的提供商越来越多，搜索相应的云服务比较麻烦，现在有网站把云服务收录起来并做了分类，以方便开发者查找，下面笔者列举两个提供这类服务的网站。

1. DevStore（http://www.devstore.cn/）

这个网站除了收集常用的开发者需要的云服务外，还收集设计、产品等相关的工具。这个网站的云服务分类界面如图 10-7 所示。

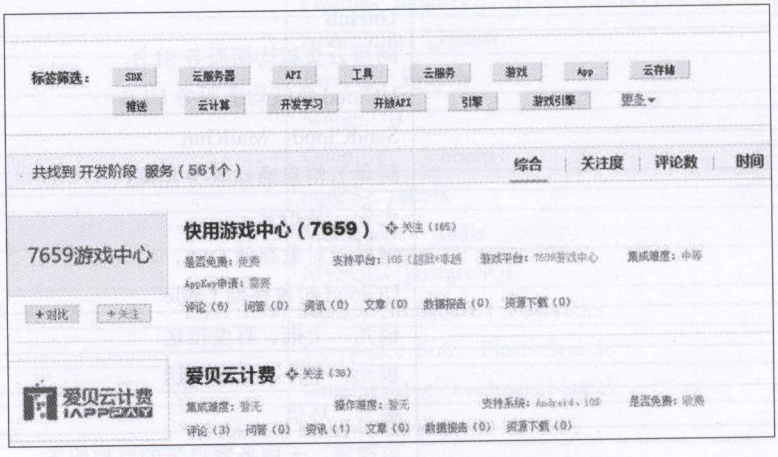


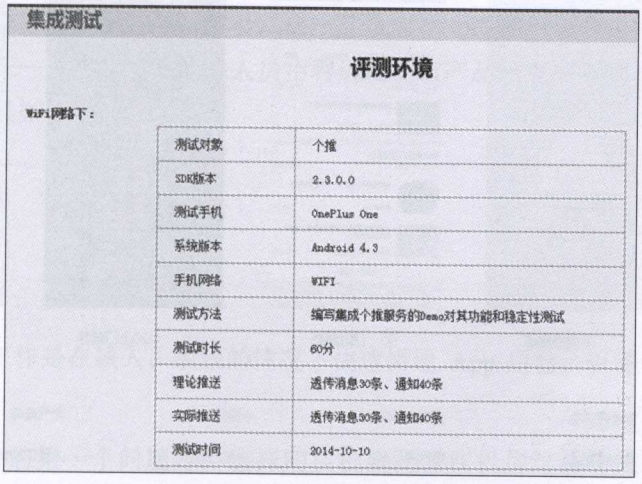
图 10-7　DevStore 的云服务分类界面

（图片来源：http://www.devstore.cn/service/newproductList/sta3.html）

另外 DevStore 网站还附带了对云服务的评测，以个推安卓版 V2.3.0.0 为例，评测报告包含下面的内容。

- 评测环境
- 集成过程

评测报告如图 10-8 所示。



集成测试

评测环境

WiFi网络下：

测试对象	个推
SDK版本	2.3.0.0
测试手机	OnePlus One
系统版本	Android 4.3
手机网络	WIFI
测试方法	编写集成个推服务的Demo对其功能和稳定性测试
测试时长	60分
理论推送	透传消息30条、通知40条
实际推送	透传消息30条、通知40条
测试时间	2014-10-10

图 10-8　DevStore 的评测报告

（图片来源：http://www.devstore.cn/evaluation/testInfo/119-196.html#332）

2. 百度 APIstore（http://apistore.baidu.com/）

百度 APIstore 的云服务分类界面如图 10-9 所示。

图 10-9　百度 APIstore 的云服务分类界面

（图片来源：http://apistore.baidu.com/astore/classificationservicelist.html）

APIstore 也提供了独特的 API 框架：开发者经过简单的几步勾选，就可以将所需云服务和工程模板打包成开发框架导入到工程中使用，如图 10-10 所示。

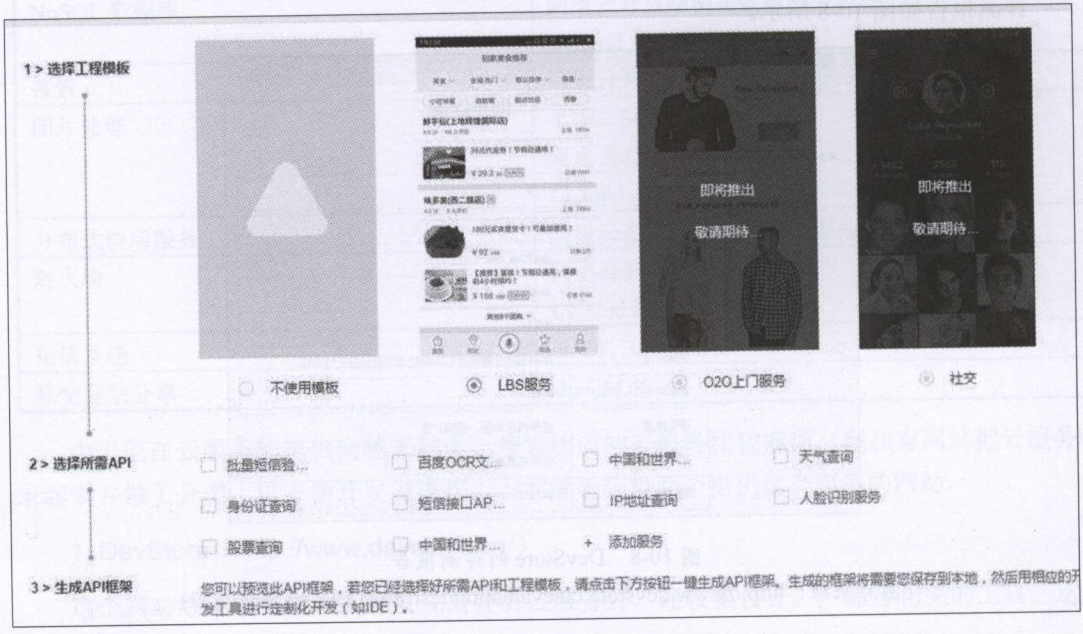


图 10-10　APIstore 的 API 框架

（图片来源：http://apistore.baidu.com/apiworks/apiframeedit）

10.3　架构的演进

App 后台的架构是由业务规模驱动而演进的，App 后台是为业务服务的，App 后台的价值在于能为业务提供其所需要的功能，不应过度设计。

从一个项目的角度，当 App 访问量不大的时候去追逐高性能 App 后台的架构是舍本逐末，这时候的主要工作是快速搭建 App 后台，让 App 尽快上线给用户提供服务，验证商业模式的正确性，同时快速迭代产品。

当 App 访问量不断飞涨，这时要在保证快速迭代的前提下，同时也兼顾高性能和高可用。

当 App 访问量增长到一定阶段后，增长曲线会有所放缓，但业务变得更加复杂，对高性能和高可用的要求也更高，性能问题、模块间的耦合、代码的复杂性会更加突出和明显，这时要使用业务拆分、分布式服务调用，甚至是技术转型等问题。

10.3.1　单机部署

单机部署适用于 App 项目刚启动时，可能产品经理脑袋里对 App 要做成什么样子只有一个模糊的想法，技术人员还是比较紧缺（笔者曾经遇到一个产品经理，为自己的产品找到了天使轮的投资，但是找不到合适的技术人员，该产品经理在 QQ 群里诉苦，每天的工作都是到处招聘人，投资人每天一个电话询问技术人员招聘进度，该产品经理一听到电话响心里就慌了），这时项目的基本情况如下。

- 缺钱
- 缺人
- 需求多变
- 时间不够用

这个阶段主要工作是在缺人、缺钱的情况下快速搭建 App 后台，以便让 App 尽快上线投入到市场。

搭建 App 后台的第一个问题是：选择购买机器托管在机房这种传统的 IDC，还是使用 UCloud 等云服务器？

在移动互联网时代，由于信息的流动速度更快了，极大可能出现一夜之间 App 就火爆起来，用户访问量飞速增长（脸萌、足记等 App 就是这种情况），应对飞涨的访问量最简单有效的方法就是升级硬件（例如升级 CPU、内存、带宽），传统的 IDC 升级的过程如下。

- 和客户经理商谈所需硬件的价格或在线选择具体的配置。
- 在线支付或银行转款。
- 确认钱到账后，等待 IDC 安排工作人员升级硬件。

在这个流程中由于需要大量人力的介入，很难做到几分钟内完成升级硬件。

使用 UCloud 云服务后，升级硬件就简单了。

- 在用户后台选择需要的硬件配置。
- 在线支付。
- 升级就完成，有的服务升级需要重启服务器。

整个过程算起来不到 5 分钟，简单、快捷，最重要的是能在最短的时间内应对访问的压力。

一个新的 App 项目也常常面临下面的问题。

- 人员问题：难找到专业的基础架构技术人员、运维管理人员。
- 时间成本问题：自行搭建基础架构和运维管理，消耗过多的时间和精力。

- 资金投入问题：项目未上线前，很难预测其用户的规模和受欢迎程度。如果前期盲目性大批量自购硬件及机房带宽，会导致前期成本投入过大、设备利用率极低、现金流占用过度的现象。

使用 UCloud 等云服务器就能一次性解决上面的问题，因此笔者是很推崇使用 UCloud 等云服务器的。

另外在这个阶段的技术选型，要结合团队自身的情况考虑。

- 团队做过什么项目？
- 团队熟悉什么技术？

在团队以往项目和熟悉技术的情况下，搭建一个能满足业务需求的最简化 App 后台架构。一般来说，这个阶段使用的是单点部署和简单化设计，App 后台极简化架构如图 10-11 所示。

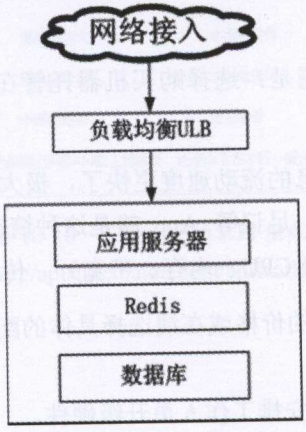


图 10-11　App 后台极简化架构

App 后台极简化架构把 HTTP 服务器、应用服务器、Redis、数据库都部署在一台云服务器。

这个架构设计有 3 个关键点需要解释一下。

1. 为什么要加入负载均衡 ULB？

一般来说，只有在访问量达到一定的程度时，才需要使用负载均衡把请求分发到集群中的服务器，减轻单台服务器的压力。

因为 UCloud 的负载均衡 ULB 是免费的，在这个阶段提前使用负载均衡 ULB，把外网的请求转发到云服务器，云服务器就不需要直接暴露在外网，增加整个架构的安全性，同时也不

会增加额外的资金支出。

开发人员需要连接云服务器的 ssh 服务，只需要在负载均衡 ULB 中把 ssh 端口的请求转发到云服务器的 ssh 端口，就能通过 ssh 操作云服务器。

2. 为什么一开始就使用 Redis？

因为 Redis 既能用作缓存，又能充当队列服务，使用同样的软件减轻运维的负担。同时其并发性能高，能在长时间内应对业务压力，非常适于初期的项目。

在这个业务时期，Redis 具有下面的作用。

（1）验证用户信息

在需要用户登录的 App 中，为了保持应用服务器的无状态，在需要验证用户信息的场景，每次 App 的请求附带用户的信息来验证用户的状态，由于每次 App 的请求都要验证信息，这种访问频次非常高的用户信息行为应该在 Redis 中进行，如图 10-12 所示。

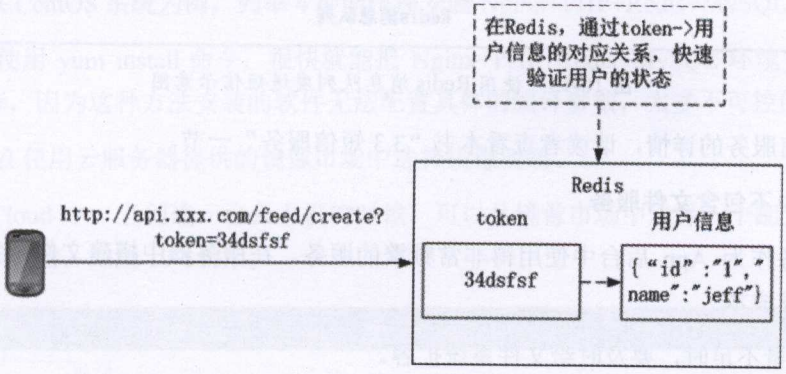


图 10-12　在 Redis 中验证用户信息示意图

关于验证用户信息的详细描述，请读者查看本书“3.1 用户验证方案”一节。

（2）充当消息队列

这个阶段架构的要求是极简化设计，但极简化设计不能牺牲用户体验。

在 App 中，有个常见的响应时间比较长的功能需求：发送验证码。

用户注册的时候，App 为了获取用户真实的社交关系，需要获取用户的手机号和手机通信录信息，获取手机号时为了验证手机号的真伪，要通过手机验证码验证手机号，但由于发送手机验证码只能通过第三方的短信发送平台发送，因此发送验证码功能有可能响应时间比较长。

为了使发送手机验证码这个功能不影响 API 接口的响应时间，通常的做法是把发送验证码

的功能放在消息队列中处理，在这里 Redis 充当消息队列的角色，如图 10-13 所示。

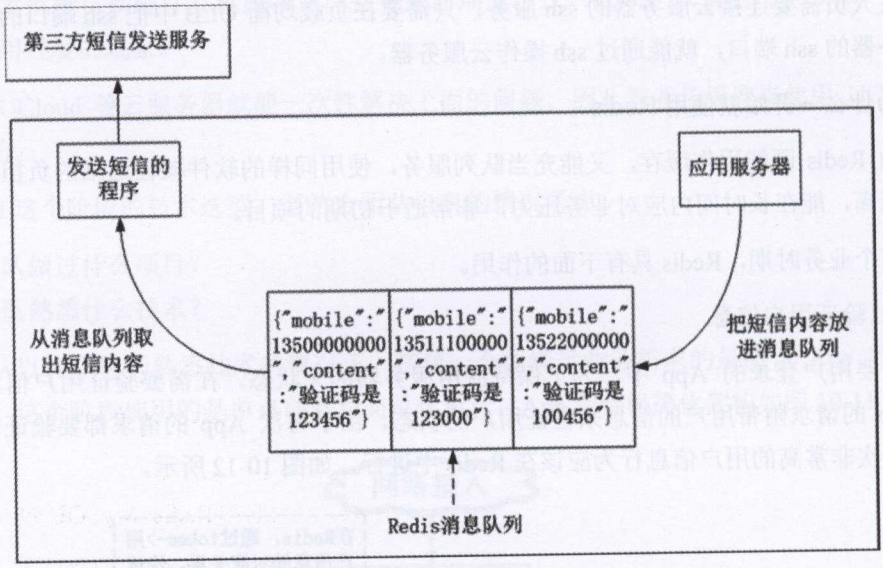


图 10-13　使用 Redis 消息队列发送短信示意图

关于短信服务的详情，请读者查看本书“3.3 短信服务”一节。

3. 架构中不包含文件服务

文件服务作为 App 后台中使用得非常频繁的服务，在服务器中搭建文件服务有下面一系列运维和开发成本.

- 当容量不足时，要及时给文件系统扩容。
- 图片的缩略、剪裁、水印等常见功能都需要研发。
- 为了保证文件服务的高可用，最少需要两台文件服务器做互备。
- 为了获得更快的文件上传和下载速度，需要更大的带宽，但带宽的成本非常高（视频网站的财务报表显示，带宽的支出占总收入的 30%以上），为了提升用户体验投入这么多的资金不划算。

开发人员在架构 App 后台、涉及文件服务的时候，都应该考虑以上的问题。

笔者一向推崇的架构原则是，**“尽量使用成熟可靠的云服务和开源软件，自身只专注于业务逻辑”**，使用文件云存储服务后以上的问题都能解决。文件云存储服务有如下的优点。

- 全网加速，根据使用场景选择最优的加速线路。
- 云端数据处理，图片缩略、剪裁、水印等常见操作都能通过相应的 API 实现，无须再

重复造轮子。

- 安全存储，多机房互相备份，保证数据安全性。
- 存储无上限、支持高并发访问。
- 按下载量计费的付费模式更加灵活，节省更多的流动资金。

因此笔者认为应该把搭建文件服务器这块从架构中移除，直接使用文件云存储服务，让研发人员投入更多的精力在业务。

在这个阶段追求的是快速搭建后台架构，在图 10-12 的架构中，如果团队的开发语言是以 PHP 为主，那么相应的架构可以是 Nginx+PHP+Redis+MySQL 这个经典的组合。

大量的初学者不知道怎样在服务器上快速安装并配置这个环境，根据笔者的观察，大部分的做法是从网络上搜索这些软件对应的安装教程（大多数是教编译安装的方法），一步一步对照着教程安装相应的软件，如果中途出现了和教程中不符合的现象，又是长时间的搜索，效率非常低。

笔者以 CentOS 系统为例，列举 4 种能快速安装 Nginx+PHP+Redis+MySQL 的方法。

（1）使用 yum install 命令，很快就能把 Nginx+PHP+Redis+MySQL 环境安装完。但这种方法不推荐，因为这种方法安装的软件无法配置具体的编译参数，太多不可控的因素。

（2）在使用云服务器提供的镜像市场中选择镜像安装。

在 UCloud 中，当创建一台云主机的时候，可以从镜像市场中选择一个配置好基础环境的镜像创建服务器，如图 10-14 所示。

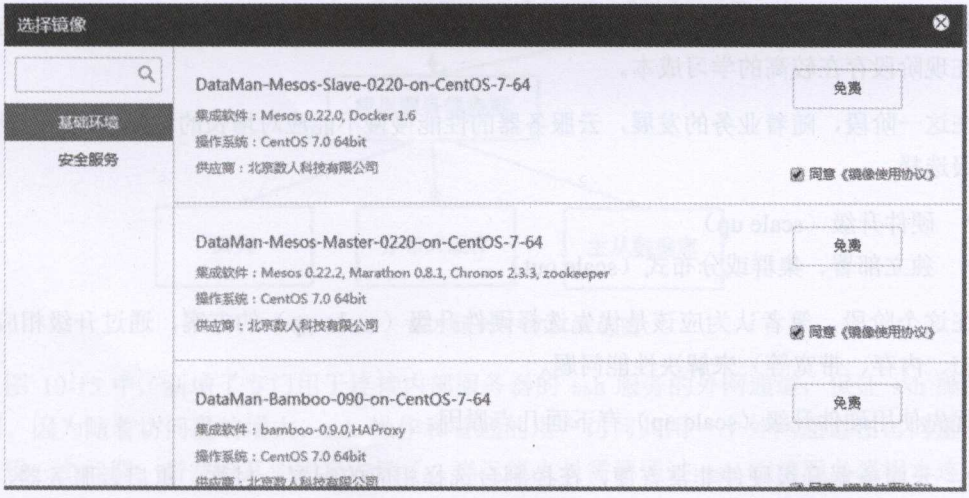


图 10-14　从镜像市场选择镜像

这种方式也是不推荐的，理由和（1）一样，安装的软件无法配置具体的编译参数，太多不可控的因素。

（3）使用一键安装包

网络上有很多一键安装包，只要输入相应的软件版本和安装路径后，能很方便地安装相应的软件，例如 lnmp 一键安装包（http://www.lnmp.org/）和 ezhttp 一键安装包（https://github.com/centos-bz/ezhttp）。

如果安装某个软件需要修改其编译参数，则查找安装脚本对应的部分修改就行。

这个阶段的项目，一般都是由开发人员来当运维人员，这类一键安装包就是帮开发者处理了常见软件的安装、配置和优化工作，搭建了大体框架，开发者如果有特殊的需求则在其基础上修改就行了，减少了运维上的负担，使用起来非常方便。

需要注意的是一键安装包为了通用性，会在系统中安装很多软件，但未必所有软件都是开发者所需要的，因此为了保证安全性，开发者在安装脚本执行完毕后还要关闭所有不需要的服务，在防火墙中严格控制允许访问的端口。

（4）Docker

Docker 是用于统一开发和部署的轻量级 Linux 容器，让开发者打包其应用及相关的依赖包到一个可移植的容器中，再发布到生产环境。通过 Docker 能保证开发者机器和生产服务器上的软件环境是一致的。

关于 Docker 的详细讲解，请读者查看本书“3.11.2 搭建一致的开发环境”一节。

新浪微博等 IT 公司也在培养开发人员用 Docker 部署环境，这种部署方式是未来的方向，虽然在现阶段存在较高的学习成本。

在这一阶段，随着业务的发展，云服务器的性能慢慢不能应对增长的访问量，这时就有两种升级选择。

- 硬件升级（scale up）
- 独立部署、集群或分布式（scale out）

在这个阶段，笔者认为应该是优先选择硬件升级（scale up）的方案，通过升级相应硬件（CPU、内存、带宽等）来解决性能问题。

优先使用硬件升级（scale up）有下面几点原因。

- 云服务器升级硬件非常方便，在控制台选择相应的配置、付款、重启云服务器，5 分钟不用就完成升级。

- 如果选择独立部署、集群或分布式（scale out），需要额外的运维成本。这是个野蛮生长的时期，产品处于不断摸索的阶段，开发人员应该把精力专注于业务和打磨产品上，尽量保持极简化架构，不应该花费太多的精力在运维。

由于每个 App 的业务不一样，可能某些 App 后台把硬件配置升级到极限也没法应付访问量上的压力，这时就要根据具体的业务情况选择独立部署、集群或分布式（scale out）这些架构升级的方案。而这些方案中，独立部署是成本最低的，集群或分布式的成本较高，优先考虑低成本的升级方案。

这个阶段的总结如下。

- 优先考虑使用云服务和熟悉的开源软件，降低人力和时间成本。
- 重视迭代速度，避免过度设计，务必先让后台能用。
- 优先考虑硬件升级（scale up）。

10.3.2　分布式部署

分布式部署阶段研发任务依然很重，架构特点是原有的单机部署架构已经不足以支撑业务的发展和高性能、高可用的要求，需要考虑某些组件独立部署、集群或分布式等架构方案，依然优先考虑云服务完成这些架构方案。在这个阶段，架构演进如图 10-15 所示。

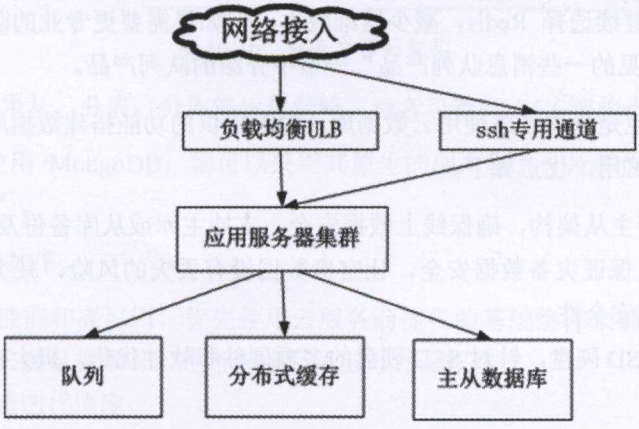


图 10-15　百万级到千万级架构图

在图 10-15 中，新增了专门用于连接内部服务器的 ssh 服务的外网通道，保证 ssh 操作随时可用。因为随着访问量的增大，ssh 操作和普通的用户访问共用一个外网通道在访问量高峰期会出现一个问题：带宽给其他用户占满了，当运维人员需要通过 ssh 连接服务器根本连不上，就算运气好勉强连入服务器，因为带宽不足，输入一个命令，要好久才有反应，甚至隔一会儿

就中断一下。如果这时服务器出了什么问题需要运维人员连接服务器紧急排查，却因为带宽不足的原因而无法连接服务器，那就没办法处理问题。当然了，不一定只在这个阶段才需要新增 ssh 通道，运维人员如果觉得有这个必要，也可以在前一阶段就新增 ssh 通道。

同时使用多台应用服务器组成的应用服务器集群，通过负载均衡 ULB 把用户请求转发到集群中的应用服务器，提供整个集群应对高访问量的能力。如果有更大的访问量，就通过在应用服务器集群添加更多的服务器应对访问量的压力，使应用服务器的负载压力不再成为系统的瓶颈。负载均衡 ULB 的高可用由云服务提供商保证，应用服务器集群的架构就具备了高可用的特性。

为了保证 Redis 缓存扩容和保证足够的 QPS，继续优先使用云内存存储 UMem 服务（兼容 Redis 协议），而不是考虑使用 Twemproxy 或 Codis 等需要大量运维成本的分布式缓存方案。云内存存储 UMem 能满足大多数应有的要求，内存空间可以在 0~500GB 之间动态调整，1GB 内存的最大 QPS（query per second）为 4000，若发现当前系统的 QPS 快到峰值，可通过升级内存提高 QPS 的上限。同时云内存存储 UMem 服务通过 3 个方面保证了高可用。

- 数据实时保存到磁盘，防止机器重启后数据丢失。
- 数据保存在一主一备两台机器中，任何一台机器上磁盘损坏数据不会丢失。
- 主机宕机，系统会自动将备机切换为主机，整个过程几秒内就能完成，无须人工干预。

队列服务可以继续选择 Redis，减少运维成本，但如果需要更专业的队列服务，读者可以选择本书“2.4.3 常见的一些消息队列产品”一节中介绍的队列产品。

主从数据库，也是优先考虑使用云数据库 UDB 提供的功能搭建数据库主从架构，其原生就支持高性能和高可用，优点如下。

- 数据库支持主从架构，确保线上数据安全。支持主库或从库备份及 7 天内多时间节点备份策略，保证灾备数据安全，让宝贵数据没有丢失的风险，减少了大量的运维成本，增加数据的安全性。
- 支持使用 SSD 硬盘，针对 SSD 硬盘的多项硬件和软件优化，极大提供了数据库的读写能力。

随着业务的发展，某些数据表的规模会以几何级增长，当数据达到一定规模的时候，查询、读取性能就下降得很厉害，数据库主从的架构不能应对业务上的读写压力，这时架构上要考虑分表，分表有两种方法。

- 水平拆分
- 垂直拆分

当业务不断发展，数据库分表后的读写性能也可能没法满足业务上的需求，这时只能采用进一步的拆分策略：分库。用 Cobar 或 MyCat 等关系型数据的分布式处理系统后，分库的架构如图 10-16 所示。

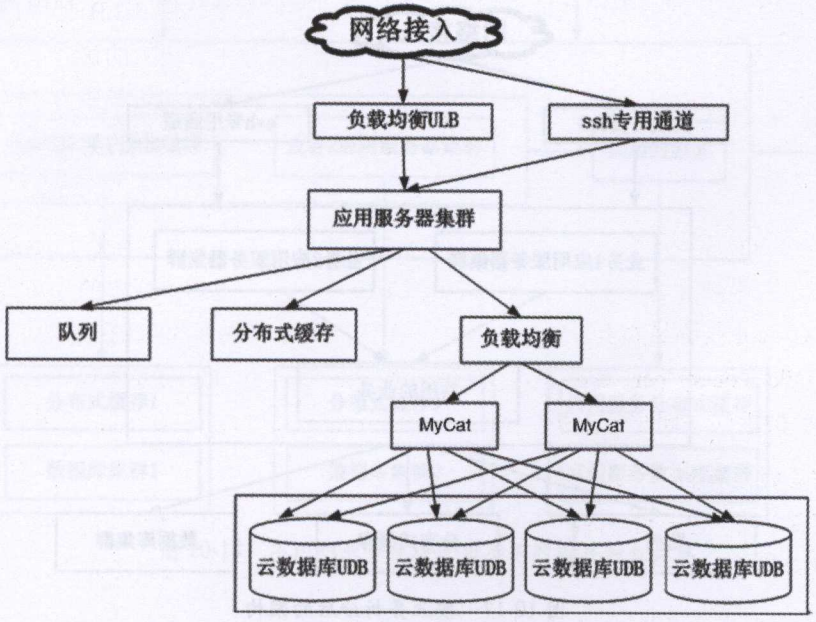


图 10-16　分表分库的架构

关于 MySQL 的主从、分表、分库等优化策略，读者可查阅“6.6 架构优化”一节。

如果数据库是使用 MongoDB，则可以采用其原生的副本集、分片等优化策略，读者可查询“8.4 高可用集群”。

这个阶段的总结如下。

- 开始考虑高性能和高可用，优先使用云服务商提供的基础组件来确保高性能和高可用，当云服务无法满足需求时，才考虑使用第三方开源软件。
- 继续保持快速迭代速度。

10.3.3　服务化

随着业务越来越复杂，App 后台聚合了大量的应用和服务，各个模块之间有很多功能重复实现，造成了开发、运维、部署的麻烦。同时，业务发展伴随着研发人员增加，代码变得更多，使用的技术和语言也会越来越多，App 后台维护成本高。

为了应对越来越复杂的业务，通过分而治之的方法把 App 后台根据业务拆分为不同的模块，各个模块之间，互相独立，功能明确。同时把一些各自模块共同的业务需求提炼为公共的服务。按业务拆分后，架构如图 10-17 所示。

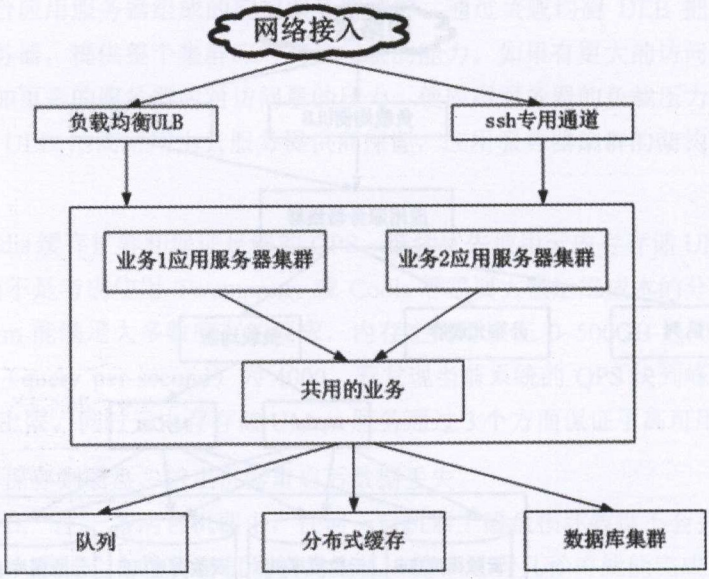


图 10-17　按业务拆分后的架构

随着业务越来越大，App 后台对读写的性能要求越来越高，所有服务器都需要和数据库以及缓存连接，在数百、数千台规模的服务器集群中，频繁的数据库读写请求有可能造成数据库连接资源不足，同时为了避免不同业务之间的相互影响（例如某个业务有慢查询导致整个数据库的性能急剧下降，从而影响到其他业务的查询），因此为了取得更好的稳定性，把缓存和数据库集群按照业务继续拆分，不同的业务使用不同的缓存和数据库，架构如图 10-18 所示。

这个阶段的总结如下。

- 架构的核心要素（高性能、高可用等）逐渐成为主角。
- 迭代速度放慢，业务渐渐趋于稳定。

注意：本节描述的是一般的 App 后台架构的发展，由于每个 App 的业务特性不一样，不一定适用于每个 App。

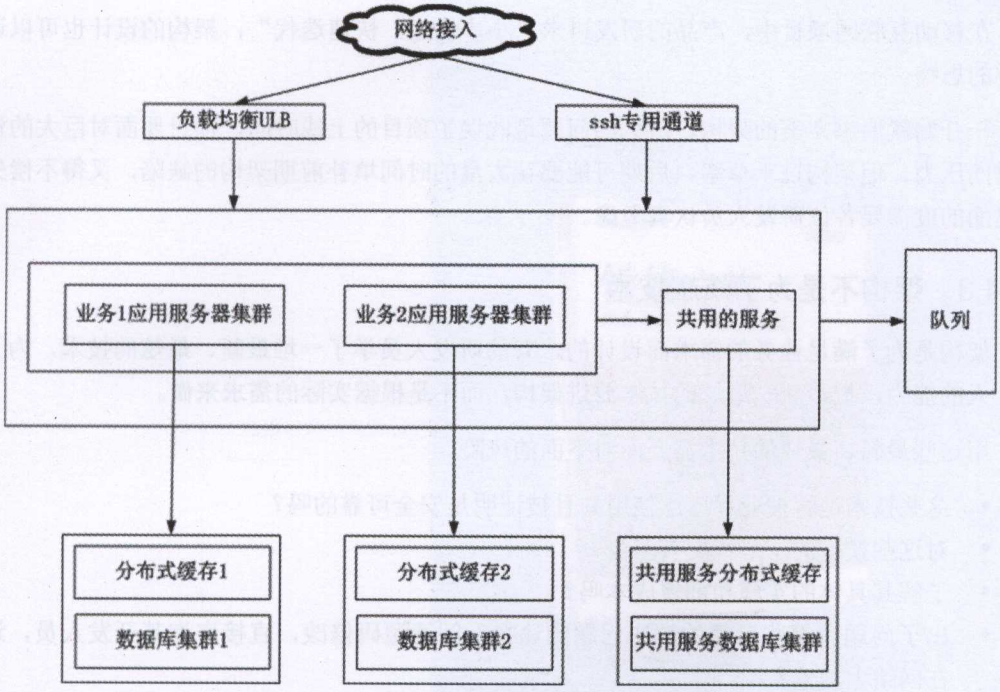


图 10-18　不同的业务模块使用不用的缓存和数据库

10.4　架构的特点

App 后台架构中有下面的几个特点需要开发者注意。

10.4.1　每个 App 的后台架构不会完全一样

每个 App 有独自的业务逻辑，遇到的问题不一样，解决方案也不一样，因此架构也不会完全相同，在 10.3 节中举例的架构演进经验是根据一般性规律总结的经验，但落实到具体的 App 后台也可能不完全适用，具体场景要具体分析。例如，有的 App 其母公司在 PC 阶段就积累了大量的用户，因此其 App 后台架构刚开始就要考虑上亿用户访问的情况，那么就不会从单机部署开始。

10.4.2　架构的演进是由业务驱动的

很多技术人员一开始就追求完美的架构，而不是根据业务的实际情况出发设计架构，就陷入了完美主义的误区。

在移动互联网项目中，产品的研发讲求“小步快走，快速迭代”，架构的设计也可以遵从同样的思路。

一开始就追求完美的架构，最大的问题是耽误了项目的上线时间，项目要面对巨大的资金和时间压力。但架构过于草率，后期可能要花大量的时间填补前期架构的缺陷，又得不偿失，这里面的度需要各位研发人员认真考虑。

10.4.3 架构不是为了炫耀技术

架构是为了满足业务的需求而设计的，有的研发人员学了一堆最新、最炫的技术，为了显示个人的能力，把最新、最炫的技术放进架构，而不是根据实际的需求来做。

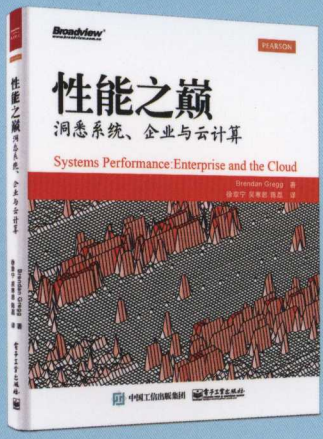
用这些最新、最炫的技术需要面对下面的风险。

- 这些技术是否被业界广泛使用并且被证明是安全可靠的吗？
- 对这些技术的了解程度有多少？
- 了解其具体的实施和部署成本吗？
- 出了问题有哪些可靠的解决问题的途径？阅读源码修改，直接咨询其开发人员，还是在网络上搜索？

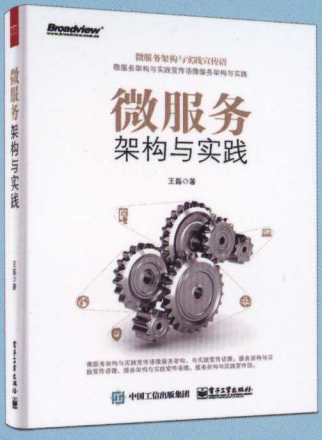
参考资料：

《大型网站技术架构核心原理与案例分析》作者：李智慧

好书分享



《性能之巅：洞悉系统、企业与云计算》
ISBN 978-7-121-26792-5



《微服务架构与实践》
ISBN 978-7-121-27591-3

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

业界力荐

作者以多年实战经验，详细阐述了后端开发，尤其是移动互联网后端开发中涉及的方方面面的技术和经验，书中推崇的“尽量使用成熟可靠的云服务和开源软件，自身只专注于业务逻辑”的理念在社会分工不断细化的今天，具有很好的借鉴意义。强烈推荐此书给希望或已经涉足后端开发、移动互联网开发和创业的朋友们，不但可补充和扩宽知识面，还有助于大家少走弯路。

何少岳 Bmob后端云CEO

随着移动互联网的快速发展，各种App应用弥漫整个市场。而为这些App提供最基本支撑的就是移动App开发技术了。本书作者从“0到1”，利用自身的项目实践经验介绍移动App后台开发架构设计和基本日常运维处理，非常适合刚踏入移动App后台开发的朋友们借鉴，让我们一起进入到这个浩瀚的领域中探索学习。

胡亚平 UCloud综合研发中心高级工程师

作者对互联网常用的几种IM协议有比较深入的讲解，书中介绍了使用版本标识的方案实现，至少收到一次消息，保证消息到达率。Gopush作为Golang推送服务器的开源实现，作者也给出不少建议和优化，在此真心感谢作者对开源软件的关注和支持。Goim作为Gopush的简化和升级版，性能更加强大，强烈推荐读者了解并投入到goim的开源社区中。

毛剑 bilibili研发总监

上架建议：计算机>网络与互联网

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

责任编辑：付　睿 @Winnie说说
封面设计：李　玲

ISBN 978-7-121-28380-2

定价：59.00元